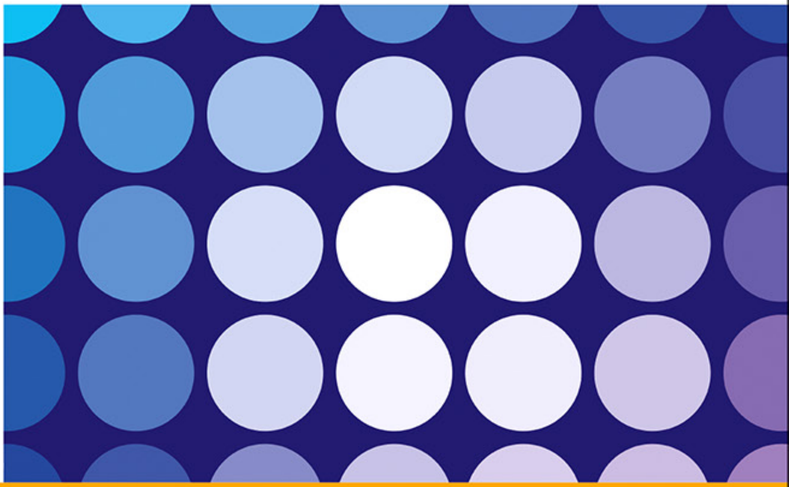
Web developme
Tags, <!docty
<title>, Attr
Semantics, HT
<input type=
Forms, Style,
Selectors, In
Cascade, Box

```
<!doctype html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Window Title</title>
   </head>
   <body>
      <h1>Heading</h1>
      <p>Standard Paragraph Text</p>
   </body>
</html>
```



HTML and CSS

The Comprehensive Guide

Jürgen Wolf

Rheinwerk
Computing

Rheinwerk Computing

The Rheinwerk Computing series from Rheinwerk Publishing offers new and established professionals comprehensive guidance to enrich their skillsets and enhance their career prospects. Our publications are written by leading experts in the fields of programming, administration, security, analytics, and more. Each book is detailed and hands-on to help readers develop essential, practical skills that they can apply to their daily work. For further information, please visit our website: www.rheinwerk-computing.com.

Philip Ackermann
JavaScript: The Comprehensive Guide
2022, 982 pages, paperback and e-book
www.rheinwerk-computing.com/5554

Sebastian Springer
Node.js: The Comprehensive Guide
2022, 834 pages, paperback and e-book
www.rheinwerk-computing.com/5556

Christian Ullenboom
Java: The Comprehensive Guide
2023, 1126 pages, paperback and e-book
www.rheinwerk-computing.com/5557

Johannes Ernesti, Peter Kaiser
Python 3: The Comprehensive Guide
2022, 1036 pages, paperback and e-book
www.rheinwerk-computing.com/5566

Bernd Öggl, Michael Kofler
Git: Project Management for Developers and DevOps Teams
2023, 407 pages, paperback and e-book
www.rheinwerk-computing.com/5555

Jürgen Wolf

HTML and CSS

The Comprehensive Guide

Rheinwerk
Computing

Dear Reader,

It's estimated that the average American spends 6–7 hours a day online. While some of that time may be spent on mobile apps, internet-connected gaming, and all the other ways we whittle away the hours, it cannot be denied how much time we spend on web pages. Web development has exploded over the last two decades and shows no signs of slowing down anytime soon.

Expert author Jürgen Wolf has set out to make sure that you have all the information you need to start building websites with HTML and CSS—and in my humble opinion, succeeded! This book has everything: detailed, step-by-step instructions, example code, practical models, a thorough grounding in the basics for beginners, and more advanced techniques for those who already have some experience. Not to mention, Jürgen provides downloadable examples and projects so that you can learn by doing.

What did you think about *HTML and CSS: The Comprehensive Guide*? Your comments and suggestions are the most useful tools to help us make our books the best they can be. Please feel free to contact me and share any praise or criticism you may have.

Thank you for purchasing a book from Rheinwerk Publishing!

Meagan White
Editor, Rheinwerk Publishing

meaganw@rheinwerk-publishing.com
www.rheinwerk-computing.com
Rheinwerk Publishing • Boston, MA

Notes on Usage

This e-book is **protected by copyright**. By purchasing this e-book, you have agreed to accept and adhere to the copyrights. You are entitled to use this e-book for personal purposes. You may print and copy it, too, but also only for personal use. Sharing an electronic or printed copy with others, however, is not permitted, neither as a whole nor in parts. Of course, making them available on the Internet or in a company network is illegal as well.

For detailed and legally binding usage conditions, please refer to the section Legal Notes.

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy:

Imprint

This e-book is a publication many contributed to, specifically:

Editor Meagan White
Acquisitions Editor Hareem Shafi
German Edition Editor Patricia Schiewald
Translation Winema Language Services, Inc.
Copyeditor Julie McNamee
Cover Design Graham Geary
Photo Credit Shutterstock: 1670310505/© Paul Aparicio; iStockphoto: 1447894700/© olaser
Layout Design Vera Brauner
Production E-Book Graham Geary
Typesetting E-Book SatzPro, Germany

We hope that you liked this e-book. Please share your feedback with us and read the Service Pages to find out how to contact us.

The Library of Congress has cataloged the printed edition as follows:
Names: Wolf, Jürgen, 1974- author.
Title: HTML and CSS : the comprehensive guide / by Jürgen Wolf.
Description: 1st edition. | Bonn ; Boston : Rheinwerk Publishing, 2023. |
Includes index.
Identifiers: LCCN 2023003226 | ISBN 9781493224227 | ISBN 9781493224234
(ebook)
Subjects: LCSH: HTML (Document markup language) | Cascading style sheets. |
Web site development.
Classification: LCC QA76.76.H94 W64 2023 | DDC 005.7/2--dc23/eng/20230127
LC record available at https://lccn.loc.gov/2023003226

ISBN 978-1-4932-2422-7 (print)
ISBN 978-1-4932-2423-4 (e-book)
ISBN 978-1-4932-2424-1 (print and e-book)

© 2023 by Rheinwerk Publishing, Inc., Boston (MA)
1st edition 2023
4th German edition published 2021 by Rheinwerk Verlag, Bonn, Germany

Contents

Preface 25

1 Introduction to the HTML Universe 31

1.1 Is This Book Even Intended for Me? 31

1.2 Different Types of Websites 32

1.2.1 Web Presence 32
1.2.2 Blog/Online Magazine/Portfolio 33
1.2.3 E-Commerce Websites: Stores without Opening Hours 35
1.2.4 Landing Page/Microsite 36
1.2.5 Web Platform: Building Your Own Social Network 36
1.2.6 Web Apps 37

1.3 Dynamic and Static Websites 37

1.3.1 Static Websites 38
1.3.2 Dynamic Websites 39

1.4 Languages for Designing and Developing on the Web 41

1.4.1 HTML: Text-Based Hypertext Markup Language 41
1.4.2 CSS: Design Language 42
1.4.3 JavaScript: Client-Side Scripting Language of the Web Browser 43
1.4.4 Server-Side Scripting Languages and Databases 43

1.5 What Do I Need to Get Started? 44

1.5.1 HTML Editor for Writing HTML Documents 44
1.5.2 Web Browser for Displaying the Website 46
1.5.3 Step by Step: Creating a Web Page and Viewing It in the Web Browser 47
1.5.4 Checking Written HTML 49
1.5.5 Good Reasons for Validating the HTML Code Anyway 52

1.6 Conventions Used in This Book 53

1.7 Summary 53

2 Basic Structure of HTML and HTML Documents 55

2.1 Syntax and Structure of HTML and HTML Documents 55

2.1.1 How to Structure a Document in HTML 55

2.1.2 Viewing the Tree Structure Using the Document Object Model Inspector 58

2.1.3 HTML Tags and HTML Elements 59

2.1.4 Nesting HTML Elements and the Hierarchical Structure 60

2.1.5 Avoiding Incorrect Nesting of HTML Elements 61

2.1.6 Omitting the End Tag of an HTML Element 62

2.1.7 Standalone HTML Tags without End Tags 63

2.1.8 Additional HTML Attributes for HTML Elements 64

2.1.9 Using Comments in HTML Documents 65

2.2 A Simple HTML Document Framework 65

2.2.1 HTML Document Type: <!doctype> 66

2.2.2 Beginning and Ending an HTML Document: <html> 67

2.2.3 Head of an HTML Document: <head> 67

2.2.4 Visible Part of an HTML Document: <body> 68

2.3 Summary 68

3 Head Data of an HTML Document 69

3.1 Overview of HTML Elements for the Head 69

3.2 <title>: Heading of the HTML Page 70

3.3 Related Topic: Naming Convention and Referencing 72

3.3.1 Valid and Good File Names for an HTML Document 72

3.3.2 Valid Directory Names and Meaningful Directory Structures 73

3.3.3 Writing a Reference to a Data Source 73

3.4 Defining the Base URL of a Web Page Using <base> 76

3.5 Referencing an External Document via <link> 78

3.6 Writing Document-Wide CSS Styles Using <style> 81

3.7 Including Scripts in Web Pages Using <script> 82

3.8 Metadata for the Document Using <meta> 85

3.8.1 The Most Commonly Used Metadata 85

3.8.2 Setting the Viewport 87

3.8.3 Specifying Useful Metadata for a Web Crawler 88

3.8.4 Useful Metadata for Search Engines 88

3.8.5 Useful Metadata for the Web Browser 90
3.8.6 Using General Metadata 91
3.8.7 My Recommendation: This Metadata Belongs in the Basic HTML Framework 92
3.8.8 HTML Attributes for the <meta> Element 92

3.9 Summary 93

4 The Visible Part of an HTML Document 95

4.1 HTML Elements for Structuring Pages 95
4.1.1 Using <body>: The Displayable Content Section of an HTML Document 96
4.1.2 Introducing the Section Elements of HTML 96
4.1.3 Using Headings with the HTML Elements from <h1> to <h6> 104
4.1.4 Creating a Header Using <header> and a Footer Using <footer> 108
4.1.5 Marking Contact Information Using <address> 110

4.2 HTML Elements for Structuring Text 111
4.2.1 Adding Text Paragraphs Using <p> 112
4.2.2 Forcing Line Breaks Using <br> 113
4.2.3 Adding Optional Line Breaks Using <wbr> 114
4.2.4 Forcing Spaces and Preventing Wrapping Using "&nbsp;" 115
4.2.5 Adding a Topic-Based Separation Using <hr> 116
4.2.6 Adding Paragraphs or Citations Using <blockquote> 117
4.2.7 Defining a General Section Using <div> 118
4.2.8 Using <main>: An HTML Element for the Main Content 120
4.2.9 Labeling Content Separately Using <figure> and <figcaption> 121
4.2.10 Creating Unordered Lists Using <ul> and <li> 122
4.2.11 Creating Ordered Lists Using <ol> and <li> 123
4.2.12 Reversing the Numbering of an Ordered List 124
4.2.13 Changing the Numbering of an Ordered List 124
4.2.14 Nesting Lists within Each Other 125
4.2.15 Creating a Description List Using <dl>, <dt>, and <dd> 128

4.3 Using Semantic HTML 130
4.3.1 HTML without a Precise Structure 130
4.3.2 Generic Structuring Using <div> 132
4.3.3 Semantic Structuring Using the Elements Provided in HTML 135
4.3.4 What's the Use of Those Semantic HTML Elements? 138

4.4 HTML Elements for Text Markups 138
4.4.1 Marking Up Abbreviations or Acronyms Using <abbr> 140

4.4.2 Marking Up Text as the Source of a Working Title Using <cite> 140
4.4.3 Marking Up Computer Code Representation Using <code> and <pre> 141
4.4.4 Keyboard Input Using <kbd> and Program Output Using <samp> 142
4.4.5 Marking Up Text as a Definition Using <dfn> 143
4.4.6 Marking Up Text as a Variable Using <var> 144
4.4.7 Changing the Text Direction Using <bdo> and <bdi> 144
4.4.8 Emphasizing Text Using <em>, <strong>, <i>, and <b> 145
4.4.9 Highlighting Text Using <mark> 146
4.4.10 Placing Text between Quotes Using <q> 147
4.4.11 Underlining or Crossing Out Text Using <u> and <s> 148
4.4.12 Marking Changes of Text Using <ins> and <del> 150
4.4.13 Displaying Text as Superscript or Subscript Using <sup> and <sub> ... 151
4.4.14 Marking Dates and Times Using <time> 151
4.4.15 Marking the Small Print Using <small> 155
4.4.16 Using <ruby>, <rp>, and <rt> for Annotations about Pronunciation ... 155
4.4.17 Grouping Ranges of Individual Text Passages Using <span> 156

4.5 Related Topic: Character Encoding 157

4.5.1 From Bytes to Character Encoding 158
4.5.2 From ASCII to ISO-8859 158
4.5.3 Beyond the Byte Boundary with Unicode 159

4.6 Character Entities in HTML 160

4.6.1 Structure of a Character Entity in HTML 160

4.7 Summary 161

5 Tables and Hyperlinks 163

5.1 Structuring Data in a Table 163

5.1.1 A Simple Table Structure Using <table>, <tr>, <td>, and <th> 164
5.1.2 Combining Columns or Rows Using “colspan” or “rowspan” 166
5.1.3 HTML Attributes for the Table Elements 169
5.1.4 Structuring Tables Using <thead>, <tbody>, and <tfoot> 169
5.1.5 Grouping Columns of a Table Using <colgroup> and <col> 172
5.1.6 Labeling Tables Using <caption> or <figcaption> 174

5.2 Electronic References (Hyperlinks) Using <a> 177

5.2.1 Inserting Links to Other HTML Documents on Your Own Website 178
5.2.2 Inserting Links to Other Websites 181
5.2.3 Opening Links with the “target” Attribute in a New Window 182
5.2.4 Email Links with “href=mailto: . . .” 183

5.2.5 Setting Links to Other Types of Content 185
5.2.6 Adding Download Links Using the “download” Attribute 186
5.2.7 Setting Links to Specific Parts of a Web Page 188
5.2.8 Creating Links to Phone Numbers 191
5.2.9 HTML Attributes for the HTML Element <a> 192

5.3 Summary 193

6 Graphics and Multimedia 195

6.1 Embedding Images Using <img> 196
6.1.1 Adding Images to an HTML Document 196
6.1.2 Specifying the Height and Width of a Graphic 200
6.1.3 Labeling Images Using <figure> and <figcaption> 202
6.1.4 HTML Attributes for the HTML Element <img> 204

6.2 Creating Link-Sensitive Graphics (Image Maps) 204
6.2.1 Use HTML Attributes for the HTML Element <area> 207
6.2.2 Referencing Defined Areas of the HTML Element <area> 208
6.2.3 HTML Attributes of <area> 208

6.3 Loading the Appropriate Image Using <picture> 210
6.3.1 HTML Attributes of <source> 211
6.3.2 Multiple Image Sources with the HTML Attribute “srcset” 212

6.4 Adding an Icon for the Website (Favicon) 213

6.5 Using Vector Graphics in HTML Documents 214
6.5.1 Adding SVG as a Graphic Reference Using <img> 215
6.5.2 Embedding SVG Directly into the Web Page Using <svg> 216
6.5.3 SVG Tags for Vector Graphics 217
6.5.4 Overview of Graphical SVG Elements 217
6.5.5 Further Notes on Using SVG 219
6.5.6 Mathematical Formulas Using MathML 219

6.6 Drawing Graphics Using <canvas> 221

6.7 Playing Videos Using the HTML Element <video> 222
6.7.1 HTML Attributes for the HTML Element <video> 224
6.7.2 Adding Subtitles to a Video Using <track> 225
6.7.3 Playing Videos via YouTube 228

6.8 Playing Audio Files Using the HTML Element <audio> 229
6.8.1 HTML Attributes for the HTML Element <audio> 230

6.9 Including Other Active Content 231
6.9.1 HTML Element <embed> 232

6.9.2 HTML Element <object> 232
6.9.3 HTML Element <iframe> 233

6.10 Summary 235

7 HTML Forms and Interactive Elements 237

7.1 Defining a Space for Forms 238

7.2 HTML Input Fields for Forms 239
7.2.1 A Single-Line Text Input Field Using <input type="text"> 240
7.2.2 A Password Input Field Using <input type="password"> 240
7.2.3 A Multiline Text Input Field Using <textarea> 241
7.2.4 A Selection List or Dropdown List Using <select> 242
7.2.5 Creating a Group of Radio Buttons Using <input type="radio"> 244
7.2.6 Adding a Text Label Using <label> 245
7.2.7 Using Checkboxes via <input type="checkbox"> 245
7.2.8 Using Fields for File Uploads via <input type="file"> 246
7.2.9 Overview of Various Buttons 247
7.2.10 Using a Hidden Input Field via <input type="hidden"> 248
7.2.11 Writing Form Fields outside of <form>...</form> 248
7.2.12 Multiple Submit Buttons for Different URLs 249

7.3 Special Types of Input Fields 250
7.3.1 An Input Field for Colors Using <input type="color"> 251
7.3.2 An Input Field for a Date Using <input type="date"> 252
7.3.3 An Input Field for a Time Using <input type="time"> 252
7.3.4 Input Fields for Date and Time 253
7.3.5 Input Fields for the Month and the Week 254
7.3.6 An Input Field for Searches Using <input type="search"> 254
7.3.7 An Input Field for Email Addresses Using <input type="email"> 255
7.3.8 An Input Field for a URL Using <input type="url"> 255
7.3.9 An Input Field for Phone Numbers Using <input type="tel"> 256
7.3.10 An Input Field for Numbers Using <input type="number"> 256
7.3.11 An Input Field for Numbers of a Certain Range 256
7.3.12 Outputting Values and Calculations Using <output> 256

7.4 The HTML Attributes for Input Fields 257
7.4.1 Setting the Input Focus Using the HTML Attribute “autofocus” 258
7.4.2 (De)activating Autocompletion Using the “autocomplete” Attribute 258

7.4.3 A List of Suggestions for Using the HTML Attribute “list” and <datalist> ... 259
7.4.4 Specifying Minimum and Maximum Values and the Step Size ... 259
7.4.5 Selecting or Entering Multiple Values Using “multiple” ... 260
7.4.6 Regular Expressions for Input Fields Using “pattern” ... 260
7.4.7 A Placeholder for an Input Field Using “placeholder” ... 260
7.4.8 Defining an Input as Required Using the “required” Attribute ... 261
7.4.9 Controlling Error Messages for Input Fields ... 261

7.5 Other Useful Helpers for Input Fields ... 263
7.5.1 Disabling Form Elements Using the HTML Attribute “disabled” ... 264
7.5.2 Permitting Read-Only for Input Fields Using the “readonly” Attribute ... 265
7.5.3 Useful Keyboard Shortcuts and Tab Sequence for Input Fields ... 265
7.5.4 Grouping Form Elements Using <fieldset> and <legend> ... 266
7.5.5 Progress Display via <progress> ... 267
7.5.6 Visualizing Values Using <meter> ... 268

7.6 Sending Form Data Using PHP ... 268
7.6.1 Transferring the Data from the Web Browser for Further Processing ... 269
7.6.2 The “POST” Method ... 271
7.6.3 The “GET” Method ... 271
7.6.4 Processing the Data Using a PHP Script ... 272

7.7 Interactive HTML Elements ... 275
7.7.1 Expanding/Collapsing Content Using <details> and <summary> ... 275
7.7.2 A Dialog Box via <dialog> ... 276

7.8 Summary ... 277

8 Introduction to Cascading Style Sheets 279

8.1 The Story of CSS ... 280

8.2 The Basic Principle of Using CSS ... 281
8.2.1 Structure of a CSS Rule ... 283
8.2.2 Declaring a Selector ... 283
8.2.3 Using Comments for CSS Code ... 284
8.2.4 A Few Notes on Formatting CSS Code ... 285

8.3 Integrating CSS into HTML ... 285
8.3.1 Style Statements Directly in the HTML Tag Using the HTML Attribute “style” ... 286

8.3.2 Style Statements in the Document Head Using the HTML Element <style> 287
8.3.3 Integrating Style Statements from an External CSS File Using <link> 288
8.3.4 Combining CSS Rules in the Head Section and in External CSS Files 289
8.3.5 Recommendation: You Should Separate HTML and CSS 291
8.3.6 Testing Alternate Stylesheets during Development 291
8.3.7 Integrating Style Statements from an External CSS File Using “@import” 293
8.3.8 Media-Specific Stylesheets for Specific Output Devices 293
8.3.9 Media-Specific Stylesheets with CSS 295

8.4 Analyzing CSS in the Web Browser 295

8.5 Summary 296

9 The Selectors of CSS 297

9.1 The Simple Selectors of CSS 298
9.1.1 Addressing HTML Elements Using the Type Selector 299
9.1.2 Addressing HTML Elements Using a Specific Class or ID 301
9.1.3 Universal Selector: Addressing All Elements in a Document 308
9.1.4 Addressing Elements Based on Attributes Using the Attribute Selector 310
9.1.5 An Attribute Selector for Attributes with a Specific Value 312
9.1.6 Attribute Selector for Attributes with a Specific Partial Value 315
9.1.7 CSS Pseudo-Classes: The Selectors for Specific Features 318
9.1.8 The Convenient Structural Pseudo-Classes in CSS 322
9.1.9 Other Useful Pseudo-Classes 329
9.1.10 Pseudo-Elements: The Selectors for Nonexistent Elements 330

9.2 Combinators: Concatenating the Selectors 332
9.2.1 The Descendant Combinator (E1 E2) 334
9.2.2 The Child Combinator (E1 > E2) 335
9.2.3 The Adjacent Sibling Combinator (E1 + E2) 337
9.2.4 The General Sibling Combinator (E1 ~ E2) 338

9.3 Recommendation: How to Use Efficient and Simple CSS 340
9.3.1 How to Write Well Performing CSS 340
9.3.2 Recommendation: Keep the CSS Code as Simple as Possible 342

9.4 Summary 343

10 Inheritance and Cascading 345

10.1 The Principle of Inheritance in CSS 345
10.1.1 Be Cautious When Using Relative Properties 349
10.1.2 Not Everything Gets Inherited 350
10.1.3 Enforcing Inheritance Using “inherit” 350
10.1.4 Restoring the Default Value of a CSS Feature (“initial”) 352
10.1.5 Forcing Inheritance or Restoring a Value ("unset") 352
10.1.6 Forcing Inheritance or Restoring Values for All Properties 353

10.2 Understanding the Control System for Cascading 354
10.2.1 The Origin of a Stylesheet 354
10.2.2 Increasing the Priority of a CSS Feature Using “!important” 355
10.2.3 Sorting by Importance and Origin 356
10.2.4 Sorting by Weighting the Selectors (Specificity) 357
10.2.5 Summary of the Cascading Rules System 361
10.2.6 Analyzing the Cascading in the Browser 362

10.3 Related Topic: Passing Values to CSS Features 363
10.3.1 Different Units of Measurement in CSS 363
10.3.2 Character Strings and Keywords as Values for CSS Features 365
10.3.3 Many Ways of Using a Color in CSS 366
10.3.4 Angular Dimensions in CSS 372
10.3.5 Passing Values via Short Notation to a CSS Feature 373

10.4 Summary 374

11 The Box Model of CSS 375

11.1 Classic Box Model of CSS 376
11.1.1 Specifying the Content Area Using “width” and “height” 376
11.1.2 Specifying the Inner Spacing Using “padding” 378
11.1.3 Creating the Border Using “border” 379
11.1.4 Setting Up the Outer Margin Using “margin” 379
11.1.5 Collapsing Margins 381
11.1.6 Determining the Total Width and Total Height of a Box 385

11.2 Newer Alternate Box Model of CSS 386
11.2.1 Using the “box-sizing” Box Model 388
11.2.2 Using the Alternate Box Model 388

11.3 Analyzing the Box Model in the Browser 392

11.4 Box Model for Inline Elements 393

11.5 Designing Boxes 393
11.5.1 Adding and Designing a Border Using the “border” Property 393
11.5.2 Setting a Background Color Using “background-color” 397
11.5.3 Using Background Images 397
11.5.4 Making Boxes Transparent 405
11.5.5 Adding a Gradient 406
11.5.6 Adding a Shadow Using the “box-shadow” Feature 409
11.5.7 Adding Round Corners Using the CSS Feature “border-radius” 411

11.6 Related Topic: Web Browser Prefixes (CSS Vendor Prefixes) 413

11.7 Summary 416

12 CSS Positioning 417

12.1 Positioning via CSS Feature “position” 417
12.1.1 Normal Positioning (“position: static”) 418
12.1.2 Positioning Elements Using “top”, “right”, “bottom”, and “left” 420
12.1.3 Relative Positioning (“position: relative”) 421
12.1.4 Absolute Positioning (“position: absolute”) 422
12.1.5 Fixed Positioning (“position: fixed”) 426
12.1.6 Sticky Positioning (“position: sticky”) 429

12.2 Controlling Stacking Using “z-index” 431

12.3 Floating Boxes for Positioning via “float” 434
12.3.1 Terminating the Float 438
12.3.2 Combining Floats into One Entity 440

12.4 Flexible Boxes of CSS 443
12.4.1 Aligning the Flexbox 443
12.4.2 Setting the Flexibility of the Flexbox 451
12.4.3 Determining the Order of the Boxes 454

12.5 Summary 454

13 Creating Responsive Layouts with CSS 457

13.1 Basic Theoretical Knowledge of Responsive Web Design 457
13.1.1 Using Specific Media Types 458
13.1.2 Media Queries for Media Features 461
13.1.3 Integrating and Applying Media Queries for Media Features 461
13.1.4 Basic Structure of a Media Feature Query 462

13.1.5 Which Media Features Can Be Queried? 464
13.1.6 Crucially Important: The Viewport for Mobile Devices 465
13.1.7 Use “em” Instead of Pixels for a Layout Break in Media Queries 469
13.1.8 Layout Breaks (Breakpoints) 471
13.1.9 No More Math Games Thanks to "box-sizing: border-box;" 472
13.1.10 What Happens to Web Browsers That Don’t Understand Media Queries? 472

13.2 Let’s Create a Simple Responsive Layout 472
13.2.1 Let’s Create the Basic Framework Using HTML 473
13.2.2 Setting General CSS Features 474
13.2.3 What Should I Use as a Basic Version without Media Queries: Mobile First? 475
13.2.4 Setting the Layout Break (Breakpoint) 480
13.2.5 Adding More Layout Breaks 482
13.2.6 Customizing the Main Content 487

13.3 Even More Flexible Elements 489
13.3.1 Use Relative Font Sizes instead of Pixels 489
13.3.2 Making Images Responsive 489
13.3.3 Flexible Images in Maximum Possible Width 493
13.3.4 Hiding Images Entirely 495
13.3.5 Loading the Right Image for the Screen Width: <picture> 496
13.3.6 Using Area-Covering Images 498

13.4 CSS Grid Layout 501
13.4.1 Creating a Grid for the Content 501
13.4.2 Placing Elements in the Grid 504
13.4.3 Layout Changes Made Easy 510
13.4.4 Spacing between Grid Lines 511
13.4.5 Checking the Grid in the Web Browser 512

13.5 Changing the Behavior of HTML Elements Using “display” 513
13.5.1 “display: block”, “display: inline”, and “display: inline-block” 513
13.5.2 Hiding Elements Using “display:none” 515
13.5.3 Further Values for “display” 516

13.6 Calculations Using CSS and the “calc()” Function 516

13.7 Summary 519

14 Styling with CSS 521

14.1 Designing Texts with CSS 521
14.1.1 Selecting Fonts via “font-family” 522

14.1.2 Providing Fonts via Web Fonts: “@font-face” ... 526
14.1.3 Using Icons via Icon Fonts ... 532
14.1.4 Setting the Font Size Using “font-size” ... 536
14.1.5 Italic and Bold Fonts via “font-style” and “font-weight” ... 543
14.1.6 Creating Small Caps Using “font-variant” ... 544
14.1.7 Defining Line Spacing via “line-height” ... 545
14.1.8 A Short Notation for Font Formatting Using “font” ... 546
14.1.9 Specifying Letter and Word Spacing via “letter-spacing” and “word-spacing” ... 547
14.1.10 Setting the Text Alignment Using “text-align” ... 548
14.1.11 Setting the Vertical Alignment via “vertical-align” ... 550
14.1.12 Indenting Text Using “text-indent” ... 551
14.1.13 Underlining Text and Striking Text Through Using “text-decoration” ... 552
14.1.14 Uppercase and Lowercase Text via “text-transform” ... 553
14.1.15 Adding Shadow to Text via “text-shadow” ... 554
14.1.16 Splitting Text into Multiple Columns Using “column-count” ... 555

14.2 Designing Lists with CSS ... 557
14.2.1 Customizing Bullet Points Using “list-style-type” ... 557
14.2.2 Using Images as Bullets via “list-style-image” ... 559
14.2.3 Positioning Bulleted Lists via “list-style-position” ... 560
14.2.4 Short Notation “list-style” for Designing Lists ... 560
14.2.5 Creating Navigation and Menus via Lists ... 561

14.3 Designing Appealing Tables with CSS ... 566
14.3.1 Creating Fixed-Width Tables ... 566
14.3.2 General Recommendation: Designing Appealing Tables with CSS ... 567
14.3.3 Collapsing Borders for Table Cells Using “border-collapse” ... 568
14.3.4 Setting the Spacing between Cells via “border-spacing” ... 569
14.3.5 Displaying Empty Table Cells Using “empty-cells” ... 570
14.3.6 Positioning Table Captions via “caption-side” ... 571

14.4 Adjusting Images and Graphics Using “width” and “height” ... 571

14.5 Transforming Elements with CSS ... 574
14.5.1 Scaling HTML Elements via “transform: scale()” ... 575
14.5.2 Rotating HTML Elements Using “transform: rotate()” ... 576
14.5.3 Skewing HTML Elements Using “transform: skew()” ... 577
14.5.4 Moving HTML Elements Using “transform: translate()” ... 577
14.5.5 Combining Different Transformations ... 578
14.5.6 Other HTML Elements ... 579

14.6 Creating Transitions with CSS ... 580

14.7 Styling HTML Forms with CSS ... 581
14.7.1 Neatly Structuring an HTML Form ... 582
14.7.2 Aligning Form Elements with CSS ... 584
14.7.3 Designing Form Elements with CSS ... 587

14.8 Summary ... 590

15 Testing and Organizing 591

15.1 Web Browser Tests: What's Possible? ... 591
15.1.1 Validating HTML and CSS ... 592
15.1.2 Which Browsers Are Visitors Currently Using? ... 592
15.1.3 CSS Web Browser Test ... 593
15.1.4 HTML5 Web Browser Test ... 594
15.1.5 Can I Use That? ... 595
15.1.6 Feature Query Using the "@supports" Rule ... 596

15.2 Viewing Websites in Different Sizes ... 596

15.3 Setting Up a Central Stylesheet ... 598
15.3.1 Combining Everything Back into One File to Shorten the Load Time ... 600

15.4 CSS Reset or Normalization? ... 600
15.4.1 Built-In Style Presets of the Web Browser and CSS Reset ... 600
15.4.2 Normalization: The Alternative to CSS Reset ... 602

15.5 Summary ... 603

16 The CSS Preprocessor Sass and SCSS 605

16.1 Sass or SCSS Syntax ... 605

16.2 From Sass/SCSS to CSS ... 606

16.3 Installing and Setting Up Sass ... 607
16.3.1 Online CSS Preprocessor without Installation ... 607
16.3.2 Setting Up Sass Using Visual Studio Code ... 608
16.3.3 Installing Sass for the Command Line ... 610

16.4 Using Variables with Sass ... 611

16.5 Nesting with Sass ... 613

16.6 Mixins ("@mixin", "@include") ... 615

16.7 Extend ("@extend") ... 618

16.8 Media Queries and “@content” 621
16.9 Operators 624
16.10 Adjusting Colors and Brightness 625
16.11 Sass Control Structures 628
16.12 Functions “@function” 632
16.13 “@import” 633
16.14 Comments 634
16.15 Summary 635

17 A Brief Introduction to JavaScript 637

17.1 JavaScript in Web Development 638
17.2 Writing and Executing JavaScript Programs 640
  17.2.1 Integrating a JavaScript File in an HTML File 641
  17.2.2 Writing JavaScript within HTML 643
  17.2.3 Position of JavaScript and Its Execution in the HTML Document 644
  17.2.4 Attributes for Manipulating the Load Behavior of JavaScript (“async”, “defer”) 645
  17.2.5 The <noscript> Element for No JavaScript 645
17.3 JavaScript Output 646
  17.3.1 Standard Dialogs (and Input Dialog) 646
  17.3.2 Outputting to the Console 647
  17.3.3 Outputting to the Website 649
  17.3.4 Running JavaScript without a Web Browser 651
  17.3.5 Annotating JavaScript Code with Comments 652
17.4 Using Variables in JavaScript 652
  17.4.1 Defining Constants 655
  17.4.2 Strict Mode Using “"use strict"” 656
17.5 Overview of JavaScript Data Types 657
  17.5.1 Number Data Type (Numbers) 657
  17.5.2 String Data Types (Strings) 658
  17.5.3 Template Strings 660
  17.5.4 Boolean Data Type 660
  17.5.5 Undefined and Null Data Types 661
  17.5.6 Objects 662
  17.5.7 Converting Data Types 662
17.6 Arithmetic Operators for Calculation Tasks in JavaScript 663

17.7 Conditional Statements in JavaScript 665
17.7.1 "true" or "false": Boolean Truth Value 666
17.7.2 Using the Various Comparison Operators in JavaScript 667
17.7.3 Using the "if" Branch 668
17.7.4 Using the Selection Operator 669
17.7.5 Logical Operators 669
17.7.6 Multiple Branching via "switch" 670

17.8 Multiple Repetitions of Program Statements via Loops 672
17.8.1 Increment and Decrement Operators 672
17.8.2 The Header-Controlled "for" Loop 673
17.8.3 The Header-Controlled "while" Loop 674
17.8.4 The Footer-Controlled "do-while" Loop 674
17.8.5 Ending the Statement Block Using "break" 675
17.8.6 Jumping to the Start of the Loop via "continue" 675

17.9 Summary 676

18 Arrays, Functions, and Objects in JavaScript 677

18.1 Functions in JavaScript 677
18.1.1 Different Ways to Define a Function in JavaScript 678
18.1.2 Calling Functions and Function Parameters 680
18.1.3 Return Value of a Function 683
18.1.4 The Scope of Variables in a Function 683
18.1.5 Defining Functions in Short Notation (Arrow Functions) 686
18.1.6 Using a Function in a Web Page 687

18.2 Arrays 689
18.2.1 Accessing the Individual Elements in the Array 690
18.2.2 Multidimensional Arrays 691
18.2.3 Adding or Removing New Elements in an Array 692
18.2.4 Sorting Arrays 697
18.2.5 Searching within Arrays 698
18.2.6 Additional Methods for Arrays 699

18.3 Strings and Regular Expressions 700
18.3.1 Useful Functions for Strings 700
18.3.2 Applying Regular Expressions to Strings 701

18.4 Object-Oriented Programming in JavaScript 702
18.4.1 What Exactly Are Objects? 702
18.4.2 Creating Objects via Constructor Functions 703
18.4.3 Creating Objects via the Class Syntax 704

18.4.4 Accessing the Object Properties and Methods: Setters and Getters 705
18.4.5 The Keyword “this” 708

18.5 Other Global Objects 709
18.5.1 The Top Object “Object” 709
18.5.2 Objects for the Primitive Data Types: Number, String, and Boolean 709
18.5.3 “Function” Object 711
18.5.4 “Date” Object 711
18.5.5 “Math” Object 711
18.5.6 “Map” Object 712
18.5.7 “Set” Object 712

18.6 Summary 713

19 Changing Web Pages Dynamically 715

19.1 Introduction to the DOM of an HTML Document 715

19.2 The “document” Object 717

19.3 DOM Programming Interface 717

19.4 Accessing Elements in the DOM 718
19.4.1 Finding an HTML Element with a Specific “id” Attribute 719
19.4.2 Finding HTML Elements with a Specific Tag Name 720
19.4.3 Finding HTML Elements with a Specific “class” Attribute 723
19.4.4 Finding HTML Elements with a Specific “name” Attribute 724
19.4.5 Using “querySelector()” and “querySelectorAll()” 725
19.4.6 Other Object and Property Collections 727

19.5 Changing an HTML Element, an Attribute, or the Style 730
19.5.1 Changing the Content of HTML Elements Using “innerHTML” 730
19.5.2 Changing the Value of an HTML Attribute 732
19.5.3 Changing the Style (CSS) of an HTML Element 733

19.6 Responding to JavaScript Events 735

19.7 Handling the Events Using the Event Handler 736
19.7.1 Setting Up an Event Handler as an HTML Attribute in the HTML Element 737
19.7.2 Setting Up Event Handlers as a Property of an Object 737
19.7.3 Setting Up an Event Handler via “addEventListener()” 738

19.8 Overview of Common JavaScript Events 740
19.8.1 The JavaScript Events of the UI (Window Events) 740
19.8.2 JavaScript Events That Can Occur in Connection with the Mouse 742

19.8.3 JavaScript Events for Devices with a Touchscreen 744
19.8.4 JavaScript Events That Occur in Connection with the Keyboard 744
19.8.5 JavaScript Events for HTML Forms 745
19.8.6 JavaScript Events for the Web APIs 745

19.9 More Information about Events with the “event” Object 745

19.10 Suppressing the Default Action of Events 748

19.11 The Event Flow (Event Propagation) 749
19.11.1 More about the Bubbling Phase 750
19.11.2 Canceling Bubbling via the “stopPropagation()” Method 751
19.11.3 Intervening in the Event Flow during the Capturing Phase 752
19.11.4 Additional Information on the Capturing and Bubbling Phases 753

19.12 Adding, Changing, and Removing HTML Elements 754
19.12.1 Creating and Adding a New HTML Element and Content 755
19.12.2 Targeting HTML Elements Even More Exactly in the DOM Tree 756
19.12.3 Adding a New HTML Element Even More Targeted to the DOM Tree 760
19.12.4 Deleting an Existing HTML Element from the DOM Tree 762
19.12.5 Replacing an HTML Element in the DOM Tree with Another One 763
19.12.6 Cloning a Node or Entire Fragments of the DOM Tree 764
19.12.7 Different Methods to Manipulate the HTML Attributes 765
19.12.8 The <template> HTML Tag 768

19.13 HTML Forms and JavaScript 770
19.13.1 Reading Text Input Fields with JavaScript 771
19.13.2 Reading Selection Lists with JavaScript 772
19.13.3 Reading Radio Buttons and Checkboxes with JavaScript 773
19.13.4 Intercepting Buttons with JavaScript 775
19.13.5 Controlling the Progress Indicator <progress> with JavaScript 776

19.14 Summary 777

20 An Introduction to Ajax 779

20.1 An Introduction to Ajax Programming 779
20.1.1 A Simple Ajax Example in Execution 781
20.1.2 Creating the “XMLHttpRequest” Object 783
20.1.3 Making a Request to the Server 783
20.1.4 Sending Data 784
20.1.5 Determining the Status of the “XMLHttpRequest” Object 785
20.1.6 Processing the Response from the Server 787

20.1.7 The Ajax Example during Execution 787
20.1.8 A More Complex Ajax Example with XML and DOM 788
20.1.9 The JSON Data Format with Ajax 793

20.2 Summary 797

The Author 799

Index 801

Service Pages I
Legal Notes II

Preface

The first questions you'll probably ask yourself about a book of this scope is whether it's a book for you at all and what you'll learn from it. The title already indicates that HTML and CSS are covered here. If you've flipped through the book a bit or skimmed the table of contents, you'll have noticed that it contains much more than HTML and CSS. Just a few years ago, an author could leave it at simply writing a book on HTML with a little CSS. Then, with the new standard HTML version, the demands for creating websites have increased.

The focus of the book is still on HTML and CSS. Thus, in the first seven chapters, you'll get to know the basics of HTML. Since HTML is the basic language for website development, this book is also attractive to newcomers because it starts from scratch. Even if you're still familiar with the old HTML school (i.e., you've already dealt with HTML before the time of HTML5), you should approach this book as a newcomer and definitely read through the first seven chapters to give yourself an update of your probably outdated HTML knowledge.

Web design and the layout of websites are nowadays implemented via *Cascading Style Sheets* (CSS), which are described very extensively in nine chapters of this book. While the book doesn't intend to be a replacement for pure CSS books or web design books, you'll definitely learn many important and useful basics about web design and website layout here. If you're interested in this and haven't had any experience with it yet, you'll find this book a great companion to start with.

Because many of the innovations of the HTML standard can be addressed via *JavaScript* (application programming interfaces [APIs]), it's obvious that JavaScript must also be treated as a web programming language. In this context, you should be aware that you'll only get a brief and simple introduction to JavaScript, which is necessary to at least use and understand the Document Object Model (DOM) manipulations in practice. The scope of JavaScript alone would fill an entire book. I mention this here only to avoid raising any false hopes. In addition, if you've never dealt with a programming language before, JavaScript will probably be your first real programming language. But if you already do have experience with another programming language, JavaScript won't cause you any problems.

What's not covered in this book are web programming languages such as PHP, Python, or Java. While PHP is used in a few examples in the book, it's only used in passing to demonstrate specific examples to you. For web programming with PHP and MySQL, you should definitely get other literature if you want to dive deeper into it. However, a prerequisite for PHP and MySQL for programming dynamic websites is again a good knowledge of HTML and CSS, which is another good reason to read and work through this book. So, if you're drawn to dynamic web programming, this book is an ideal first building block for that.

Website or Web Page?

A *web page* is a single page of an internet site. The *website*, on the other hand, is the complete internet presence. As a rule, therefore, a website usually consists of several individual web pages. I'll explain these two terms here right at the beginning so that you understand the difference when we talk about a web page or website, because it tends to cause confusion.

Book Resources

All listings from the book are available for you to download from the website for this book: *www.rheinwerk-computing.com/5695*. Click on the **Resources** tab. You'll see the downloadable files along with a brief description of the file content. Click the **Download** button to start the download. Depending on the size of the file (and your internet connection), it may take some time for the download to complete.

HTML5 and the "Living Standard"

To avoid misunderstandings about HTML as a hypertext markup language, HTML5, and HTML as a living standard, these terms will be clarified at the beginning of the book. At the beginning, I briefly referred to HTML5. After the invention of the web, there were different phases in which a standard for HTML was created. Initially, only the W3C as a body took care of the standardization of HTML, which hasn't always been smooth with regard to the browser manufacturers. As the web was rapidly moving from a pure hypertext system to a platform for web applications, the W3C attempted to promulgate a new XML-based standard, XHTML, although browser vendors preferred to continue development based on HTML.

As a result, Apple, Opera, and Mozilla founded a new group, Web Hypertext Application Technology Working Group (WHATWG), which was then joined by Google and Microsoft to continue working on the HTML standard independently of the W3C. When the W3C wasn't able to enforce the change to XML, the focus returned to the further development of HTML. This gave rise to the HTML5 standard together in cooperation with the WHATWG. Yet, the harmony between W3C and WHATWG didn't last long. While the W3C stuck to versioning, the WHATWG wanted to make it a living standard without a number.

Again, however, the W3C has joined the WHATWG, and, as of May 28, 2019, HTML is a living standard without a version number. Accordingly, there will probably never be an HTML6. Related information on this topic is available at the following:

- *https://html.spec.whatwg.org*: HTML living standard
- *https://www.w3.org*: The World Wide Web Consortium (W3C)

Target Group

HTML and CSS are the main focus of the book as they cover about 85% of the entire scope. The remaining 15% of the book is dedicated to somewhat more complex but also essential topics such as JavaScript, DOM manipulations, and Ajax—the very topics or technologies you're going to encounter sooner or later when creating websites and for which at least basic knowledge will be useful. If you count yourself among the following groups, this book can definitely be an asset to you:

- **Newcomers**
  Due to its didactic structure, the book will provide you with a comprehensive introduction to the world of HTML, CSS, and a little bit of JavaScript.
- **Returners**
  You already had the pleasure of working with an older HTML version (e.g., HTML 4.01) and want to refresh your old knowledge base? Then this book is also for you. In any case, you should read through the chapters on HTML, because the way of using HTML for websites has changed a tiny bit. Besides, the days of bringing HTML into play for styling, layout, and color are well and truly over as nowadays only CSS is used for those purposes. And if you still remember JavaScript as a gimmick or a language for rascal pranks, this book will prove you wrong.
- **Web authors, bloggers**
  If you're a web author or blogger and use HTML and a little CSS for your daily work, this book will provide you with a companion to maintain your web pages on a web-based level and structure the content properly. Even if you only use ready-made content management systems (CMS), good HTML and CSS skills are an advantage.
- **Frontend, backend, and full stack developers and programmers**
  As a developer and programmer, it's hard to avoid dealing a little with web applications, for example, to output data in the web browser as an HTML document. Of course, this depends on the programming language. If you're purely interested in web development with PHP and databases such as MySQL, a good knowledge of HTML and CSS is almost a must.

How Should I Read through the Book?

The structure of the book is very didactic, and the topics build on each other, so you should be able to handle it well as a newcomer if you work through the book from cover to cover. At the same time, if you're a returner, web author, blogger, developer, or programmer, you can always jump to the topic you want to cross-read or read up on.

For you to better understand the examples, I recommend that you also at least test them in practice and experiment with them a little. Ideally, you should try to create the examples yourself. You can find all the examples used in the book on the bonus website for the book (*www.rheinwerk-computing.com/5695*) or at *https://html-examples.pronix.de/*.

It takes a bit of patience and perseverance to work through a self-study book like this. And it's not always quantity that matters, that is, reading and learning as much as possible in as little time as possible. Even though it's uncommon today, take your time when learning the new skills, and remember that it really starts after you've finished reading the book. This book will give you a good foundation to build on. But after finishing it, you'll have to gain experience in real life yourself.

Completing the book doesn't mean you're done. You shouldn't stop learning. HTML isn't a stagnant standard but is constantly evolving, with new technologies being added all the time. Stay up to date and inform yourself regularly about innovations.

Written for You, the Reader

The topics around HTML are now extremely diverse and extensive, so it isn't easy to fit the right mixture into one book. However, I think I've managed to combine an interesting collection of traditional and contemporary topics in the book. I'm aware that not everything can be described comprehensively in a book like this. Especially about JavaScript, one could write an entirely separate book. Likewise, there are topics that aren't covered at all here.

I'm also aware that such a book isn't written for self-interest, but for you, the readers. If you miss any topics that you'd like to learn more about, I'd be very happy to hear from you. The same goes for things you didn't like so much or where you think it could be done better. Even if you like this book, I'll be even more pleased with your feedback because it lets me know that I'm on the right track with this work.

Acknowledgments

A book like this isn't made by just one person, and I'm always impressed by how many people are involved behind the scenes and have work to do with it. While you're often in the spotlight as an author, many other smaller and larger gearwheels

are necessary to ultimately realize such a book. The larger cogs definitely include the editors who coordinate the entire process. For this edition, Hareem Shafi and Meagan White were at my side. Likewise, it was a great pleasure for me that Philip Ackermann took over the expert opinion for the book. Philip's experience as a software developer and web designer—he's a book author himself—put the finishing touches on the book. In addition, there's proofreading, production, layout, cover design, typesetting, and printing—tasks that many other people have taken care of. A special thanks to Elisa, for all the inspiration that pushed me to achieve my goals.

I'd like to thank all the people who have directly and indirectly contributed to this book.

Now I wish you a lot of fun and success with this book!

Jürgen Wolf

Chapter 1

Introduction to the HTML Universe

Whether you're a beginner, developer, programmer, or blogger—as a reader you'll have certain expectations of this book. This chapter is first of all about clarifying some formalities that concern (or are necessary for) this book and to elaborate on what you can expect before you start the actual practice.

You surely have already skimmed the table of contents of this book and may have noticed that its focus is on HTML and CSS. With HTML, you'll learn the markup language for creating websites, whereas with CSS, you'll learn how to design and style websites. In addition, it also covers web programming with JavaScript, which has become indispensable.

This chapter is still taking it slow, and here's what you'll learn:

- The types of websites that exist, what technologies are used for them, and the knowledge required
- The difference between dynamic and static websites
- The basic languages you should know and be able to use as a web developer
- What you need to create an HTML document and display it in a web browser
- How to check the HTML document for errors

1.1 Is This Book Even Intended for Me?

This book is aimed at *beginners* who are simply looking to create their own website or familiarize themselves with basic web technologies at first, as well as *web authors* looking for a comprehensive read on the hot topics of HTML, CSS, and JavaScript.

In addition, future *developers* and *programmers* of web applications for web templates or dynamic websites can no longer get around a sound knowledge of HTML. Even *bloggers* or *online sellers*—who often use a platform where they can enter the content in a form without any special knowledge and generate a web presentation for the viewer—can benefit from deeper knowledge to align or customize the content more neatly or, if necessary, according to their own needs.

If you don't yet know which group you want to belong to, you're at least interested in web development (otherwise, you wouldn't be holding this book in your hands). With the background knowledge around HTML, a lot of doors will open for you.

Should I Read the Chapters in Order?

For newcomers to this subject, I recommend working through the book from cover to cover. Where possible, the individual chapters in this book are structured to anticipate later chapters as little as possible. Of course, this can't be completely avoided when you explain a topic.

Returning or more experienced readers can read the chapters of the book in any order and flip to individual topics as needed. For this reason, this book can also be used as a reference work.

1.2 Different Types of Websites

At this point, I want to provide a brief overview of what common types of websites exist today and how they are created. Separating the website types isn't that easy at first because they also depend on the goal and the technology approach, and some types overlap each other. Leaving aside the technology approach, the types can be roughly divided into six categories:

- Web presence (homepage/corporate website)
- Blog/magazine/portfolio
- E-commerce website
- Web platform (social media websites)
- Landing page/microsite
- Web app

1.2.1 Web Presence

A web presence can either be a private website or a web presence of companies, associations, authorities, business people, and so on. In the case of companies, cities, and nonprofit organizations, the term *corporate website* (also *informational website*) is often used. Especially in the business world of smaller companies or self-employed people, it's good form to be present on the web with information, offers, contact options, and so on with a web address. Even in times of social networks such as Facebook, many private individuals still create and maintain their own homepage. Most of the time, you can find more details about the person and their interests there. However, at the moment, especially among younger people, the private website is going out of fashion and is being replaced by Facebook, Instagram, and Twitter. Larger companies, associations, lawyers, artists, restaurants, doctors, craftsmen, authors, and so on are often represented on Facebook in addition to a web presence. As a rule, the primary purpose of such websites is to provide information to visitors.

Required Knowledge for a Web Presence

To create private websites or web presences for companies, associations, and so on, a good knowledge of HTML, CSS, and JavaScript is useful if you want to create the website manually. Especially when it comes to the web presence of smaller companies or public figures such as artists, lawyers, and so on, the code should definitely be free of errors. As mentioned, this is predominantly only true if you create a static website. Corporate websites of companies in particular contain dynamic elements such as news or contact forms in addition to static content. Now, many companies and individuals use ready-made (dynamic) *content management systems* (CMSs) such as WordPress for their web presence. Once such a system is set up, more in-depth knowledge isn't necessarily required because web-based software is used and formatting can be implemented in a similar way to a Microsoft Office application. However, HTML and CSS knowledge is useful and helpful here as well.

1.2.2 Blog/Online Magazine/Portfolio

Blog, derived from the combination of *web* and *log*, is a website with entries that are usually sorted chronologically and separated from each other. The person who runs the blog is often referred to as a *blogger*.

Often, a blog is also the homepage of a web presence, where visitors can read the latest posts and up-to-date information about a particular topic, company, and so on. Likewise, moderated comments and discussions for and with visitors or sharing of posts on social media are possible. This category also includes so-called magazine websites, which usually also contain many current articles, photos, and videos, in addition to being informative and educational. What the magazine industry used to be, *online magazines* are now.

Here, the terms web presence and blog are often mixed up with each other. Many companies or individuals often use a ready-made system such as WordPress or Drupal for their web presence. In addition to a blog, you can also find the usual information on these websites, such as contact options, offers, and much more. However, such blog systems aren't suitable for every company. Thus, in more discreet professions such as those of lawyers and doctors, you're more likely to find a simple web presence. Many smaller businesses, such as handicraft companies or private individuals, don't have the time required to maintain a blog on a regular basis. It doesn't look good when you visit the website of a company whose last blog entry is already a year old. This makes people wonder whether the company still exists at all.

By the way, blog culture (or net culture) isn't a trivial topic that can be dealt with here in a few lines. For example, blogs can still be divided into different typologies and then again into different operators (individuals, corporations, artists, etc.). The official blog of a company, for example, is referred to as a *corporate blog* (corporate website). Even Twitter has coined its own term with *microblogging*.

However, blogs and online magazines essentially differ from web presences or corporate websites in that they not only inform visitors about the company or the individual but also regularly present new and relevant content with added value. To create a blog or magazine website, you have two options: install blog software on a server or web space, or use a ready-made hosted solution. Installing blog software such as WordPress on a server or web space is much more flexible because here you can extend the blog with many more existing modules and templates. If there's no suitable module available, you can program one yourself.

This category also includes *portfolio websites* for designers, photographers, artists, and creatives, where they can present their work visually. This often involves installing website themes with minimalist designs for blog software (e.g., WordPress). The amount of text is often significantly reduced on such websites.

What Are PHP and MySQL?

PHP is a scripting language whose syntax is similar to that of the C programming language and is mainly used for creating dynamic websites and web applications.

MySQL is a relational database management system that's mostly used for dynamic web presences on the internet in connection with the Apache web server and the PHP scripting language.

For the installations, however, certain requirements must be met on the server or web space (e.g., access to PHP and a MySQL database), and a basic knowledge of this is an advantage if things don't work out right away with the installation. With a hosted solution such as *www.blogger.com* or *www.tumblr.com*, you don't have to bother much about this and can usually start blogging right away after a quick signup and template selection.

Required Knowledge for a Blog

Here, too, knowledge of HTML, CSS, and JavaScript is advantageous to be able to take various fine details into your own hands. CSS knowledge especially is extremely helpful because you can often change the complete web design with it. Generally, the posts of such blogs are created using web-based software. This is a web application that runs in the browser and is usually quite easy to use, like an office application for text creation. With such blog systems, you only have to worry about the content. The layout, saving, adding, and archiving of blog articles is done for you by the blog system. If you're already a developer and familiar with PHP and MySQL, for example, or if you want to learn programming in the future to write your own modules, you'll definitely need more in-depth knowledge of HTML, CSS, and JavaScript.

1.2.3 E-Commerce Websites: Stores without Opening Hours

As online shopping is becoming increasingly popular, it's no surprise that many companies want to be represented with a web store. The advantages are quite obvious: open around the clock, less personnel costs, no costs and rent for the store and the facility, and a couple more reasons.

In practice, ready-made software is used for a web store because it requires much less effort to update or maintain the product catalog, for example. Even more importantly this web store software has already been tried and tested many times and is therefore much more secure, which is particularly important when it comes to the payment process.

Thanks to ready-made web shop software, such an online store can be set up quite quickly by anyone. However, there's a long list of legal requirements here that you must follow strictly for the store to be legally valid. This starts with the obligation to publish legal data, general terms and conditions (GTC) must be present, the cancellation policy mustn't be missing, correct information on delivery time and prices and much more. If you're a layman setting up an online store, you might still want to consider a lawyer for advice.

Depending on the functionality of the web shop, software can be quite expensive (e.g., for an online store). In this context, you need to assess what's worthwhile for you. The solutions range from complete solutions offered by hosting providers to professional web store software for installation on a server or web space. Here, prices vary from free to five-figure amounts. Often, specialized software such as Shopify, Magento, or WooCommerce is used for this purpose.

Required Knowledge for a Web Store

The web store is usually operated via an access-protected user interface (UI; usually via the web browser), which is similar to a CMS for a blog. For this reason, the same applies here as for a blog: knowledge of HTML and CSS isn't absolutely necessary, but it's an advantage if you want to present the product in a better way.

If, on the other hand, you want to create an e-commerce website yourself, then in addition to HTML, CSS and JavaScript, working knowledge of a server-side language such as PHP or Ruby on Rails is necessary. The handling of databases must also be mastered here.

It isn't suitable and doesn't make sense for everyone to set up and open their own web store right away. This depends on what you want to sell and the size of the business. For those who only want to sell a handful of products and are new to the e-commerce world, it may be sufficient to offer their products at *www.ebay.com*, for example. You should keep in mind that once the web store has been set up and a lot of money has been invested, you first need visitors to your online store. However, a visitor alone is far from being a buyer.

1.2.4 Landing Page/Microsite

A *landing page* usually consists of only one web page, which is aimed at a specific goal of having visitors perform a specific action (*call to action*). This would be, for example, starting a test phase for a product, buying a product, or simply contacting us. The goal of such a landing page is to present visitors with all the elements of a product on one page so that they become potential customers. Furthermore, such pages are highly optimized for search engines in order to reach targeted audiences via social media campaigns or search engine advertising.

Often the term *microsite* is also used as a synonym for a landing page, but that isn't quite correct, as a microsite is rather an informational website, which consists of a few pages and deals exclusively with a specific topic. This concept is frequently used by companies to specifically promote a single product on a separate domain, rather than placing the product within an extensive corporate website.

Required Knowledge for a Landing Page/Microsite

You can theoretically create a landing page/microsite using HTML and CSS. But here, too, there are web construction kits, special plug-ins, and themes for a CMS available that do all the work for you. However, JavaScript technologies such as React or Angular also represent viable solutions for developing a landing page/microsite.

1.2.5 Web Platform: Building Your Own Social Network

Web platform can be used generically for the other types of websites. I use it here for websites that registered users not only can read but also to which they can add their own content online via a web browser. The functionality is often provided by a CMS. Typical social networking platforms such as Facebook, Myspace, and so on or wiki software (e.g., as used by *https://en.wikipedia.org*) are also included. Particularly in the commercial sector, such platforms can achieve much better customer-oriented support and, in smaller to larger companies, also a fruitful exchange of experience and knowledge beyond departmental boundaries.

The basic idea of such a web platform is usually that the content is enhanced with texts, images, graphics, and more through the collaborative work of registered users to provide a collection of useful information. Even if the content is created by other users, a moderator is indispensable for managing and reviewing the content.

Required Knowledge for Web Platforms

The same applies here as before for the web store and the blog. In addition, the required knowledge depends on whether you're a user or a moderator of such a web platform. With HTML knowledge, you can better structure the content to your own liking and design it using CSS. However, that depends on the platform you use. Some platforms allow the use of HTML elements only under certain conditions. If you plan to develop your own web platform, the knowledge of HTML alone is no longer sufficient. Then more extensive knowledge of development in a server-side web programming language such as PHP, Ruby, Python, or JavaScript technologies (now also possible server-side) such as React or Angular is required.

1.2.6 Web Apps

Web apps are basically ordinary web applications that resemble desktop applications. These are internet applications with many interaction options, such as you would find in an ordinary desktop application. Such applications don't necessarily have to run in a web browser. The benefits of such applications over classic web applications include improved usage and, with appropriate technology, faster performance.

Required Knowledge for Rich Internet Applications

In the past, external technologies in the form of third-party plug-ins, such as Flash Player, Java Virtual Machine (JVM), Silverlight, AIR, and Flex, were the preferred solutions for creating such web apps. Meanwhile, web apps can also be created using classic web technologies such as HTML, CSS, JavaScript, and Ajax without any plug-ins. Ready-made HTML/JavaScript-based frameworks and libraries, such as Angular, React, Ext JS, and Google Web Toolkit, are available for this purpose.

1.3 Dynamic and Static Websites

In the preceding section, we often referred to the terms content management systems (CMSs) and blog systems. Well-known representatives of such systems include WordPress, Joomla!, Drupal, Contao, and TYPO3. Once such systems have been installed on a server or web space, hardly any further knowledge is required in principle, but, as always, it's still pretty useful. CMSs are run on the server side, are programmed with modern web programming languages (mostly PHP, Ruby, and Python), and often also require a server-side database (e.g., MySQL or PostgreSQL). This assumes that appropriate resources (PHP, Ruby, Python, and/or MySQL) are available on the server and may be used. Such CMSs create dynamic websites. For this purpose, the difference between static and dynamic websites should first be explained.

1.3.1 Static Websites

In a static website, all content (e.g., text and image information) is stored unchangeably in individual files on the web server. The content of such a file is created using HTML. When you make changes to static websites, the file in question usually needs to be changed manually on the local machine and then uploaded back to the web server. The use of static websites is therefore likely to be worthwhile for smaller web presences where changes are needed relatively infrequently.

Potential *advantages* of static websites include the following:

- The cost of web hosting is cheaper because no special features such as databases or scripting languages are needed. Note, however, that the professional features no longer cost a fortune with the larger web hosts.
- Page load and load time may be faster because the page can be returned immediately from the web server in response to the request.
- Developing static websites can be easier and less expensive. However, this depends on the scope of the project and your skills.

Possible *disadvantages* of static websites include the following:

- A good knowledge of HTML is required to update the website. If you plan to create a web presence for someone using static web pages, you should be aware that you'll have to make the changes yourself most of the time.
- The initial creation of many individual files for the static website can become very time-consuming.
- Changing the design of the website can be quite expensive. In the worst-case scenario, you need to change every single file. Ideally, however, the web design of a static website is based on CSS, so only this CSS file would need to be changed.

In Figure 1.1, you can see a simplified representation of how a static web page is returned. Here, the web browser first sends a *request* for a web page to the web server that hosts the website. The web server finds this page and sends it back to the web browser as a *response*. If this web page isn't found on the web server, it returns an error message (usually with error code 404) stating that the resource couldn't be found on the server.

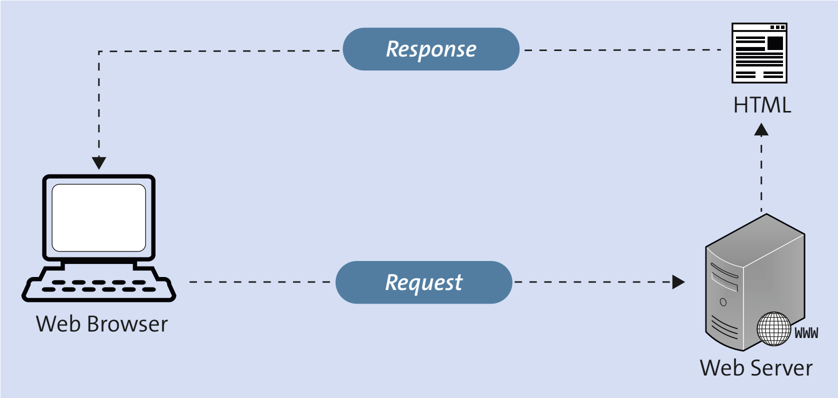


Figure 1.1 Request from the Web Browser and Return of a Static Web Page Stored on a Web Server

1.3.2 Dynamic Websites

For dynamic websites, a CMS usually generates the web pages. This usually involves keeping the content, such as text and images, separate from the technical elements, such as the layout or scripts. When a visitor visits the website, the content and technical elements on the web server are read from a database and dynamically assembled into a web page before being returned to the visitor.

In any case, such a CMS must be installed and reside in a web server environment where, depending on the CMS, different scripting languages (e.g., PHP or Python) and mostly databases (e.g., MySQL or PostgreSQL) must be present before you can install/use the CMS.

Potential *advantages* of dynamic websites include the following:

- Updating and adding new content can be done much faster via a web-based UI. As a rule, you no longer have to bother about data storage (where and how).
- Design modifications and design changes can be made in one central location. Often there are many ready-made *templates* available. Design changes affect all existing web pages at the same time.
- Such systems can be maintained without HTML and other programming skills and can even be managed by several people. New functionalities can be added at any time thanks to many existing modules/plug-ins (e.g., search feature, sitemap, online store, and forum).

Possible *disadvantages* of a dynamic website include the following:

- Web hosting incurs higher costs due to the need for special features such as scripting languages and databases. However, the costs are no longer significantly higher than for a static website, which they still were a few years ago.

- If you need to create your own or special modules or plug-ins, knowledge of programming with scripting languages becomes necessary. This could make the development take a little longer and be more expensive.

Figure 1.2 shows a very simplified representation of how a web page is dynamically generated. A web browser makes a request for a web page to a web server by entering a web address. The web server searches for and finds the page and then passes it to the application server. The application server searches the found page for commands and completes the web page. Additionally, statements for a database query can be included here. In this case, such a *query* to the database (more precisely, the database driver) is started. The database driver then returns the requested record (also called *recordset*) to the application server, where this data is inserted into the web page. The dynamic web page thus created on the web server gets sent to the web browser as a response.

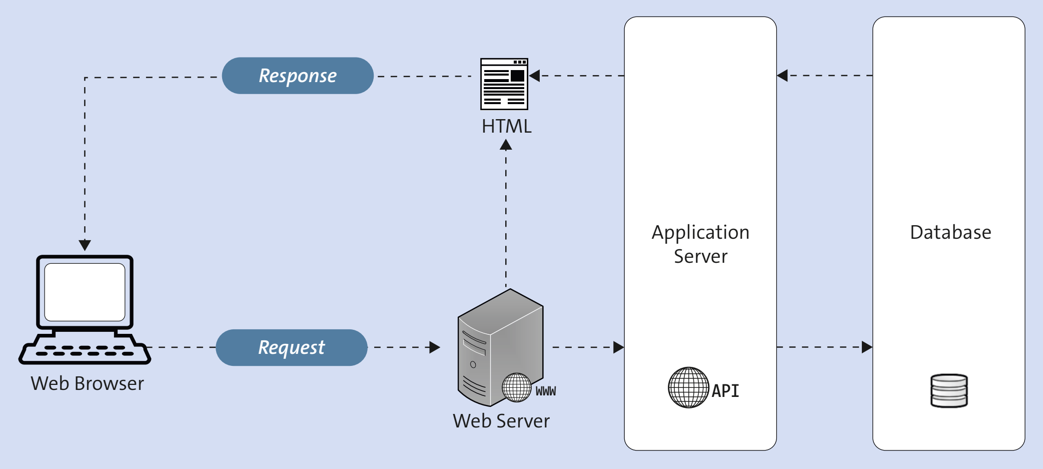


Figure 1.2 Simplified Representation of How a Web Page Is Assembled and Returned after a Web Browser Request on the Web Server

Web Server, Application Server, Database

Admittedly, I've thrown around a lot of terms here, so they should be explained briefly. While you won't have much to do with dynamic websites in this book, it can still be helpful to know these terms a little better.

- **Web server**
  This is a computer on which web server software (and usually nothing else) is installed. Such a web server is typically used to make documents available locally, on an intranet, or over the internet, as well as to transmit them to clients such as a web browser. The most important web servers are probably the *Apache HTTP Server* and *Microsoft Internet Information Services* (IIS). By the way, the location of the web server can be anywhere in the world.

- **Application server**
  An application server provides an environment for client-server applications and a web server. For example, in a web application, the web browser represents the client part of the application. The application server provides certain services (e.g., access to databases and authentication).
- **Database**
  A database is used to store a large amount of data as efficiently as possible and make it available on demand. Usually, a database consists of a database management system and the data itself. The management system of a database takes care of the structured storage and accesses to the data. Database systems provide their own database language for managing and querying the data. There are many different database systems, with MySQL and PostgreSQL currently having the largest market shares on the web.

1.4 Languages for Designing and Developing on the Web

Now that we've covered the different types of websites, this section gives you an overview of the languages you need to know as a web developer and will learn about in this book:

- **HTML**
  You'll use this language to create the content of a website.
- **CSS**
  You'll use this language to create the layout of the website.
- **JavaScript**
  You'll use this language to program the behavior of the website.

1.4.1 HTML: Text-Based Hypertext Markup Language

HTML is a purely text-based markup language for the structured representation of text, graphics, and hyperlinks in HTML documents. HTML documents can be displayed by any web browser and are therefore also considered the basis for the internet. Because HTML is a purely text-based language (*plain text format*), HTML documents can be edited and saved with any text editor. In addition to the data displayed in the web browser, an HTML document can contain other meta information. Occasionally you hear that websites are programmed with "HTML commands." However, it's wrong to speak of HTML as a programming language because in a programming language, certain tasks are solved by a sequence of commands. HTML doesn't have any commands or statements, but instead uses *markers* (also referred to as *tags*). These markers are used to structure the individual sections of an HTML document. Even though there will still be talk of "programming HTML," you should try to remain technically correct here because HTML isn't programmed, but written.

Meta Information

Meta information (or metadata) is data that's not usually displayed but contains information about the characteristics of the data (in this case, the HTML document), such as the language or author of the document.

1.4.2 CSS: Design Language

Although over the years elements were added to HTML that dealt with the visual design of a document, fortunately the decision was made to separate structuring and layout by defining them with CSS. Thus, in common practice, HTML is used only for the logical structuring of web pages.

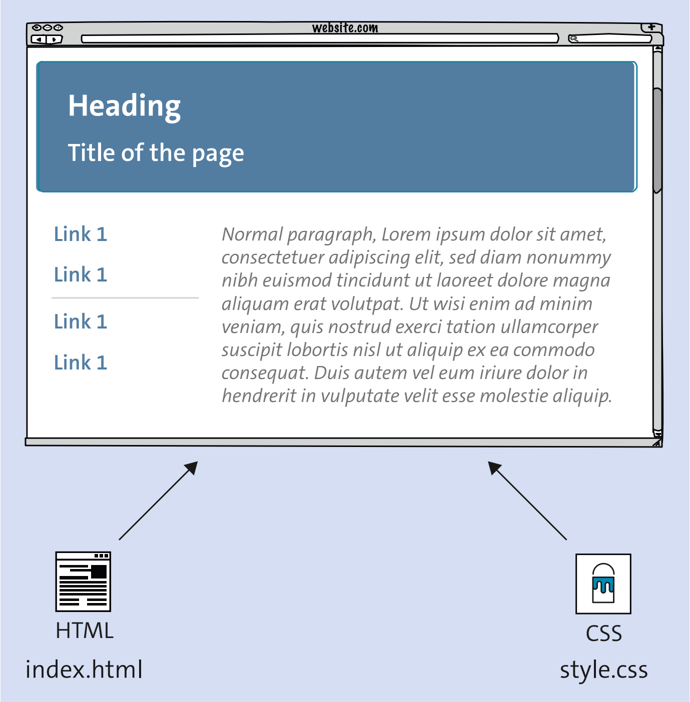

website.com
Heading
Title of the page
Link 1
Link 1
Link 1
Link 1
Normal paragraph, Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie aliquip.
HTML
index.html
CSS
style.css

Figure 1.3 Usually, HTML Code for Semantic Structuring Is in One File, and the CSS Code for Styling and Laying Out Is in Another

It's of course possible to display HTML in the browser without CSS specifications. For example, if you use heading text in the `h1` element (e.g., `<h1>heading</h1>`), this text is displayed larger than the rest of the text in the HTML document between `<h1>` and `</h1>`. If you use HTML elements for tables, they are visibly structured as a table. Texts can also be displayed in bold or italics in HTML. The way in which these HTML elements (without CSS) are ultimately displayed in the web browser is determined by the web browser provider. The HTML specification only contains recommendations on what such a default setting might look like in the browser. HTML is only used to semantically structure an HTML document with the HTML elements and should be used exclusively for this purpose in practice. CSS is used for layout and styling.

By separating HTML and CSS, content and layout are separated from each other. In practice, the HTML code (for structuring) is usually in one file, and the CSS code (for formatting and styling) in another. If you apply this separation consistently to all web pages, you can change the complete layout of all web pages with just one CSS file. The HTML files don't need to be changed. For this reason, you ideally control the visual presentation and styling of the HTML elements via CSS. CSS can be edited with a plain text editor just like HTML.

1.4.3 JavaScript: Client-Side Scripting Language of the Web Browser

Client-side scripting languages are referred to as web-based scripts that are executed on the local computer, usually by the web browser. In common practice, JavaScript is the most significant client-side scripting language. JavaScript allows you to extend the limited capabilities of HTML with user interactions, for example, to evaluate, dynamically modify, and create content. Nowadays, no modern website can do without JavaScript.

Unfortunately, JavaScript has been abused for all sorts of mischief in the past, so the reputation of this programming language hasn't necessarily been the best. In addition, there was a browser dispute between Netscape and Internet Explorer, in which Microsoft wanted to push through its own JavaScript language, JScript.

By now, the tide has turned in favor of JavaScript on one hand due to the World Wide Web Consortium (W3C) with the introduction of a *Document Object Model* (DOM), which was gradually adopted by web browser manufacturers, and on the other hand to the many *JavaScript frameworks*. Much of the innovation in standard HTML has nothing to do with ordinary HTML elements, but rather with *JavaScript APIs*.

Even if you get to know JavaScript in this book primarily as a way to extend HTML and CSS, you can already find this language outside of web browsers for mobile applications and desktop applications or on servers or microcontrollers.

1.4.4 Server-Side Scripting Languages and Databases

Server-side scripting languages aren't directly covered in this book, but should be mentioned briefly anyway because only with server-side scripting languages can web pages be generated dynamically. I've already briefly explained dynamic websites in Section 1.3.2. Well-known and common server-side scripting languages are PHP, Python, and

Ruby. These scripting languages can be used to implement blogs, forums, form mails, guestbooks, and wikis, for example. Most larger websites today are equipped with server-side techniques and often use a database connection to MySQL or PostgreSQL as well.

Many larger blogs or CMSs such as WordPress, Drupal, TYPO3, Contao, and Joomla! are based on such server-side scripting languages with database connectivity. Most of the time, these systems are based on PHP as the scripting language and MySQL as the database.

It can be quite helpful if you have basic knowledge of a scripting language such as PHP to be able to create form mails, guest books, and surveys for smaller web presences, for example. Dealing with databases such as MySQL is also quite useful. Once you're familiar with HTML, CSS, and JavaScript after reading this book, nothing can stop you from moving on to a scripting language such as PHP and a database such as MySQL to dive even deeper into web development.

1.5 What Do I Need to Get Started?

Beginners often wonder what is needed to create web pages and learn HTML. Basically, you wouldn't need anything at all because everything is on board your operating system by default. Strictly speaking, you only need a plain text editor to create web pages and a web browser to display them.

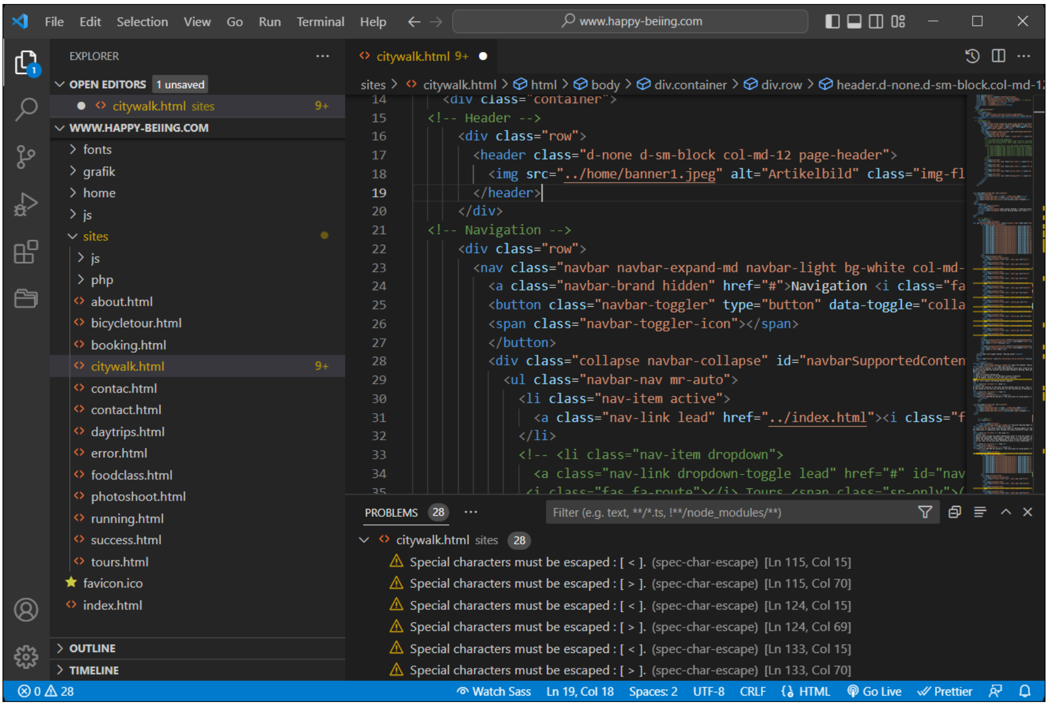
1.5.1 HTML Editor for Writing HTML Documents

As a text editor, you could theoretically use the editor installed on the system. For Microsoft Windows, this is Microsoft Editor (original name: Notepad). The TextEdit editor on the Mac also gets along splendidly with HTML. For Linux systems, the default editor depends on the distribution used. Often gedit is used here, which is also best suited for creating HTML pages.

In practice, hardly any ambitious web developer uses the operating system's standard editors; instead, they use real HTML editors (or at least universal text editors). The advantage of such special HTML editors is that you have syntax highlighting and many other helpful features at hand for creating web pages. There are a lot of free and commercial HTML editors on the market. Office word-processing programs such as Microsoft Word are less well suited, if at all, for creating pure HTML code because they often add unnecessary ballast to the HTML source code (when the files are saved in HTML format).

If you haven't decided on a particular editor yet or maybe don't quite know what you want to use, here are my brief recommendations, all of which are available for Windows, Linux, and macOS (free of charge):

- **Visual Studio Code** (*https://code.visualstudio.com*)
  The editor comes from Microsoft and has become the standard tool for many web developers. It's also my editor of choice and makes a developer's life much easier with countless extensions and language support.
- **Adobe Brackets** (*https://brackets.io*)
  Brackets was designed by Adobe as a community project purely for web application development.
- **Sublime Text** (*https://sublimetext.com*)
  Before there were countless editors on the market, Sublime Text was often the preferred editor for web developers. However, Sublime Text isn't free of charge, even though you can test this editor without any time limit.


File
Edit
Selection
View
Go
Run
Terminal
Help
www.happy-beiing.com
EXPLORER
OPEN EDITORS
1 unsaved
citywalk.html sites
WWW.HAPPY-BEIING.COM
fonts
grafik
home
js
sites
js
php
about.html
bicycletour.html
booking.html
citywalk.html
contac.html
contact.html
daytrips.html
error.html
foodclass.html
photoshoot.html
running.html
success.html
tours.html
favicon.ico
index.html
OUTLINE
TIMELINE
citywalk.html 9+
sites > citywalk.html > html > body > div.container > div.row > header.d-none.d-sm-block.col-md-1
PROBLEMS 28
Filter (e.g. text, **/*.ts, !**/node_modules/**)
citywalk.html sites 28
0 ⚠ 28
Watch Sass
Ln 19, Col 18
Spaces: 2
UTF-8
CRLF
HTML
Go Live
Prettier

```
 <div class="container">
<!-- Header -->
  <div class="row">
   <header class="d-none d-sm-block col-md-12 page-header">
    <img src="../home/banner1.jpeg" alt="Artikelbild" class="img-fl
   </header>
  </div>
<!-- Navigation -->
  <div class="row">
   <nav class="navbar navbar-expand-md navbar-light bg-white col-md-
    <a class="navbar-brand hidden" href="#">Navigation <i class="fa
    <button class="navbar-toggler" type="button" data-toggle="colla
    <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedConten
     <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
       <a class="nav-link lead" href="../index.html"><i class="f
      </li>
      <!-- <li class="nav-item dropdown">
       <a class="nav-link dropdown-toggle lead" href="#" id="nav
```

```
⚠ Special characters must be escaped : [ < ]. (spec-char-escape) [Ln 115, Col 15]
⚠ Special characters must be escaped : [ > ]. (spec-char-escape) [Ln 115, Col 70]
⚠ Special characters must be escaped : [ < ]. (spec-char-escape) [Ln 124, Col 15]
⚠ Special characters must be escaped : [ > ]. (spec-char-escape) [Ln 124, Col 69]
⚠ Special characters must be escaped : [ < ]. (spec-char-escape) [Ln 133, Col 15]
⚠ Special characters must be escaped : [ > ]. (spec-char-escape) [Ln 133, Col 70]
```

Figure 1.4 Visual Studio Code from Microsoft Is the Editor I Prefer to Use in My Daily Work

WYSIWYG Editor

What You See Is What You Get (WYSIWYG) editors are also available. With these editors, you can format and "click together" a web page virtually like with an office program for word processing. The WYSIWYG editor generates ready-to-use HTML code in the form of a file. The best-known representatives are Dreamweaver from Adobe or Google Web Designer (*https://webdesigner.withgoogle.com*). Such editors are certainly helpful if you want to work quickly or if you have more experience, but they are less suitable for learning HTML at first, even if these programs also have a text editor available. However, these environments require some training time and good web development skills before you can effectively design websites with them.

1.5.2 Web Browser for Displaying the Website

To view the HTML document created in the HTML editor of your choice, you need a web browser. As a website developer, you shouldn't settle for just one web browser, but use as many as possible for testing, as there are many small differences between different web browsers and their respective versions. It's also advisable to view a website on different devices. When you view modern websites on devices with different screen sizes, such as a desktop computer, a laptop, a tablet, and a smartphone, you'll notice that they often display differently. This is because such websites ideally adapt to the environment in which they are displayed. This adaptability is called *responsive web design*. The adjustment doesn't happen automatically, but it's the responsibility of the web designer. I'll go into greater detail about this separately in this book.

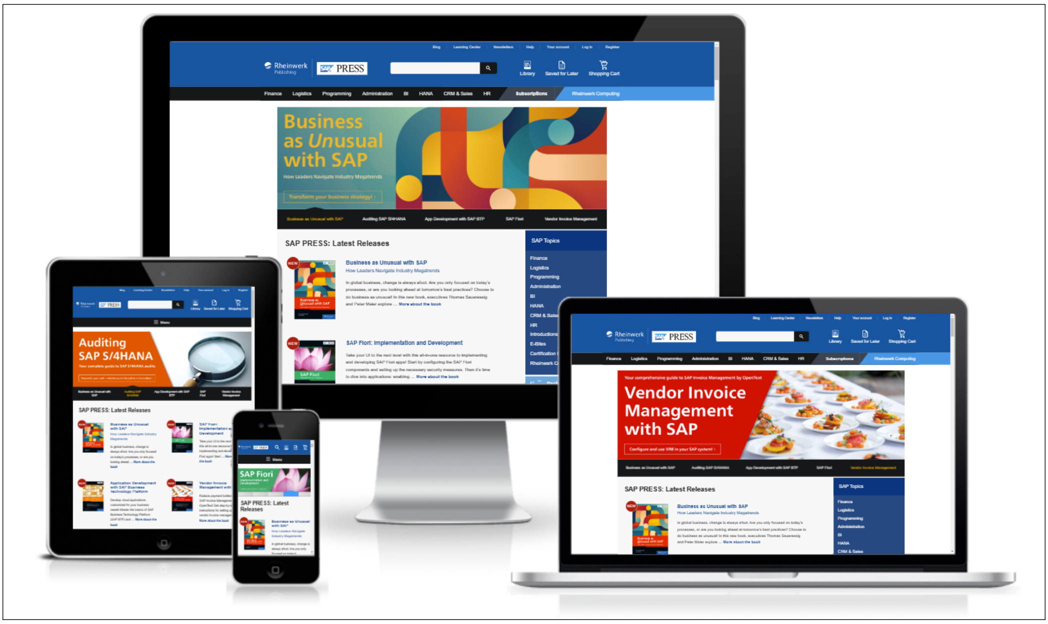

Rheinwerk Publishing
SAP PRESS
Business as Unusual with SAP
SAP PRESS: Latest Releases
Auditing SAP S/4HANA
SAP Fiori
Vendor Invoice Management with SAP

Figure 1.5 The Same Website Is Tested Here on "https://ui.dev/amiresponsive" for Different Devices

The main web browsers are currently Google Chrome, Mozilla's Firefox, Apple's Safari, and Microsoft's Edge, with Google Chrome currently having the largest market share. There are also many other browsers, such as Vivaldi, Opera, and Brave, but they only have a small market share. The manufacturers often supply their own browsers for mobile devices as well. For example, the Samsung internet browser is particularly strong on Samsung devices.

The heart of any browser is the HTML renderer (often called the browser engine), which converts (*renders*) the source code coming from the web server into a visible web page. The HTML renderers current at printing time are listed in Table 1.1.

Renderer	Browser
Quantum	Firefox
WebKit	Safari and all web browsers on iOS
Blink	Chrome, Edge, Samsung Internet, Vivaldi, Opera, Brave, and so on

Table 1.1 Different Web Browsers and the HTML Rendering Engine They Use

The fact that many browser vendors use Blink, provided by Google, as their HTML renderer makes things a little easier for you when it comes to testing: You can assume that a web page that looks good in one browser with Blink as the renderer will usually look good in the others. The same is true for Apple's WebKit.

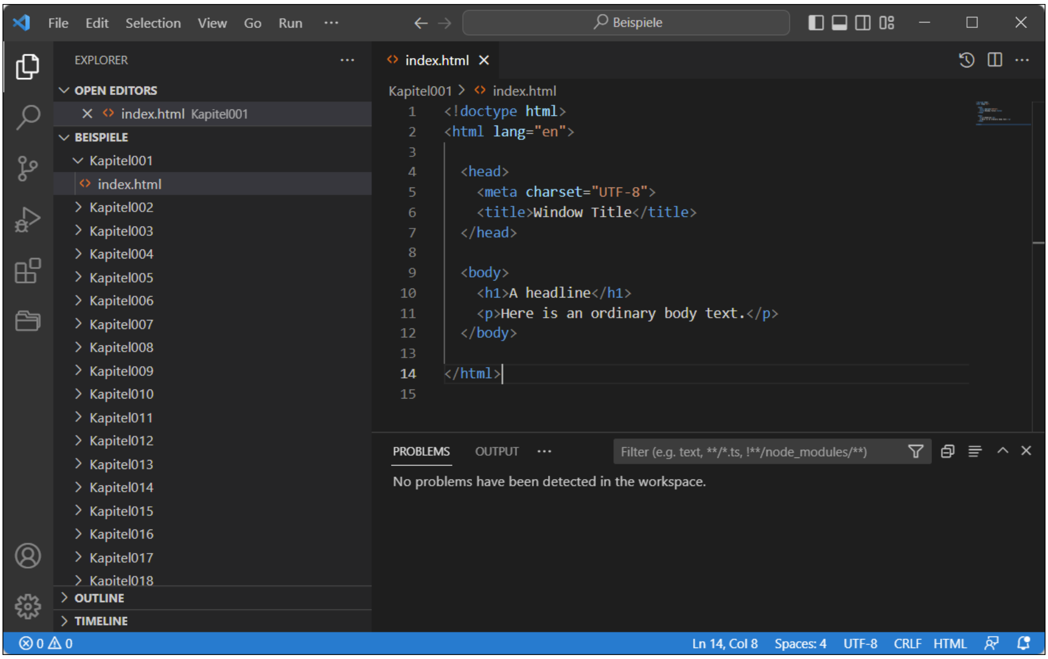
1.5.3 Step by Step: Creating a Web Page and Viewing It in the Web Browser

To get you prepared for the rest of the book, I'll now show you in four steps how to create a single web page using an editor (of your choice) and view it in your web browser.

1. Open a text or HTML editor (Section 1.5.1), and create a new empty text document. Mostly this should be accessible via the menu path **File • New File**.
2. Type the HTML code into the editor. For demonstration purposes, the following basic structure is used for this example (don't worry about the meaning of the individual lines yet):

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Window Title</title>
  </head>
  <body>
    <h1>A headline</h1>
    <p>Here is an ordinary body text.</p>
  </body>
</html>
```

Listing 1.1 /examples/chapter001/index.html


File
Edit
Selection
View
Go
Run
Beispiele
EXPLORER
OPEN EDITORS
index.html Kapitel001
BEISPIELE
Kapitel001
index.html
Kapitel002
Kapitel003
Kapitel004
Kapitel005
Kapitel006
Kapitel007
Kapitel008
Kapitel009
Kapitel010
Kapitel011
Kapitel012
Kapitel013
Kapitel014
Kapitel015
Kapitel016
Kapitel017
Kapitel018
OUTLINE
TIMELINE
index.html
Kapitel001 > index.html
PROBLEMS
OUTPUT
Filter (e.g. text, **/*.ts, !**/node_modules/**)
0 0
Ln 14, Col 8
Spaces: 4
UTF-8
CRLF
HTML

```
<!doctype html>
<html lang="en">

  <head>
    <meta charset="UTF-8">
    <title>Window Title</title>
  </head>

  <body>
    <h1>A headline</h1>
    <p>Here is an ordinary body text.</p>
  </body>

</html>
```

No problems have been detected in the workspace.

Figure 1.6 Here I've Written the HTML Code in Microsoft's Visual Studio Code Editor on Windows

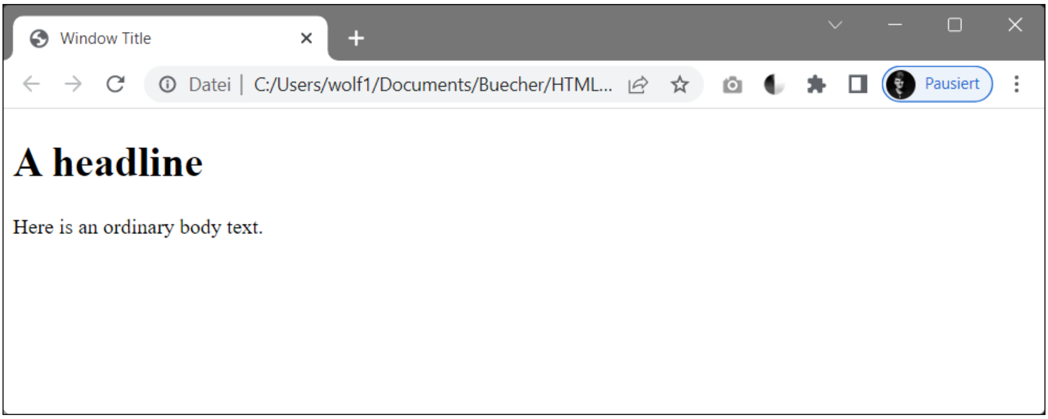
3. Save the HTML code. Ideally, you want to create a separate directory for this purpose. Most editors provide the **File • Save** or **File • Save As** command. There are two things to consider here, namely the file extension and the encoding of the file: you must save this web page with the *.htm* or *.html* file extension and make sure that the file is really saved as a *plain text* file, that is, without any vendor-specific formatting. Concerning the encoding, UTF-8 is always a good choice (but you can also use ANSI at first). With most editors, you don't have to worry about this and often find the file extension available for selection.

Using the ".htm" or ".html" File Extension

The web browser doesn't care whether you use *page.htm* or *page.html* as the file extension. The fact that there are two names at all has historical reasons that go back to the DOS world, where file could only be named according to the 8+3 rule, that is, eight characters for the file name and three characters for the file extension. As already mentioned, it's entirely up to you which file extension you use. However, I recommend that you commit to one version and always use it in the future. I've chosen to use the *.html* extension in this book.

Depending on the system, the HTML document with the extension *.html* or *.htm* is displayed in the file browser with a corresponding icon of the installed default web browser.

4. Double-click the file to view it in the web browser.



A headline

Here is an ordinary body text.

Figure 1.7 The Saved HTML Document index.html in Google Chrome on Windows

1.5.4 Checking Written HTML

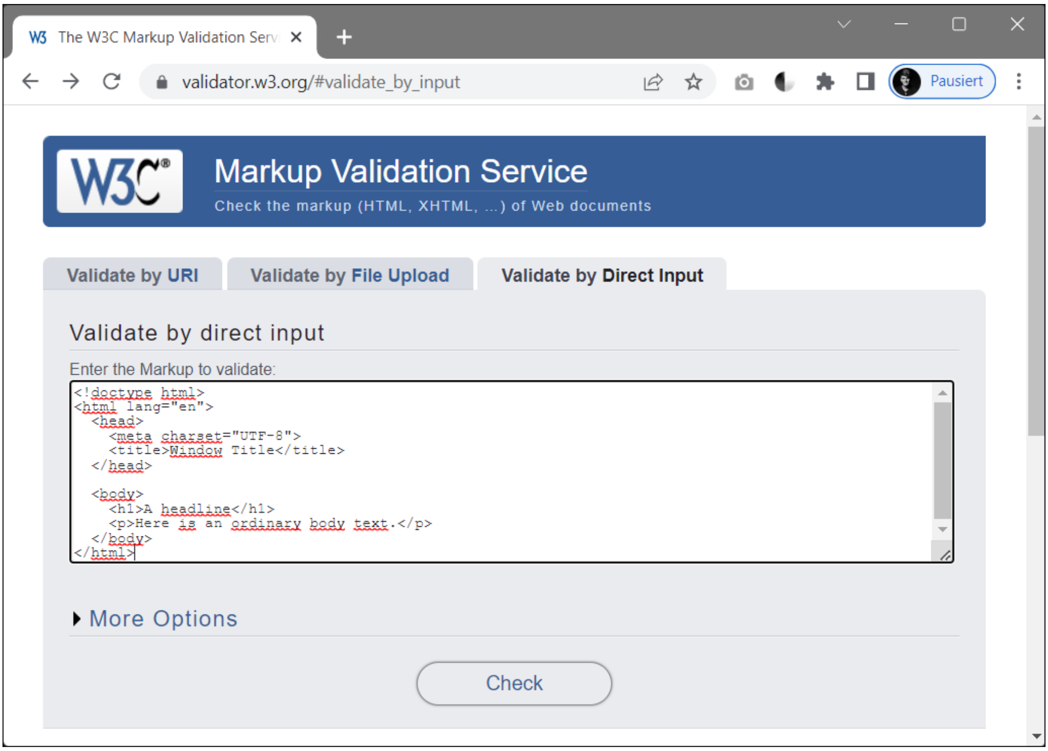
To check whether the HTML code is correct and to learn from mistakes, it's worth validating the HTML code or the web page. The easiest way to do this is to use the online tool from the W3C, which you can find at *http://validator.w3.org*.

Validation with Editors and IDEs

In many HTML editors or integrated development environments (IDEs), functions for validating HTML are often already available or can be integrated subsequently as an extension. For example, Visual Studio Code provides the extension HTMLHint for this purpose.

On this website, you can validate an existing web page (**Validate by URI**), upload an existing HTML document (**Validate by File Upload**) and have it validated, or simply copy and paste an HTML code (**Validate by Direct Input**) and then validate it.

Because you'll probably still be testing simple HTML documents on the local computer in the beginning, uploading or simply copying and pasting is a good option. In the example, the latter will be briefly demonstrated (see Figure 1.8). For this reason, you should select **Validate by Direct Input** (or *http://validator.w3.org/#validate_by_input*), copy the HTML code you entered in the editor to the clipboard, and paste it into the text box under **Enter the Markup to validate**.


W3 The W3C Markup Validation Serv ×
validator.w3.org/#validate_by_input
Pausiert
W3C®
Validate by URI
Validate by File Upload
Validate by Direct Input
More Options
Check

Markup Validation Service

Check the markup (HTML, XHTML, ...) of Web documents

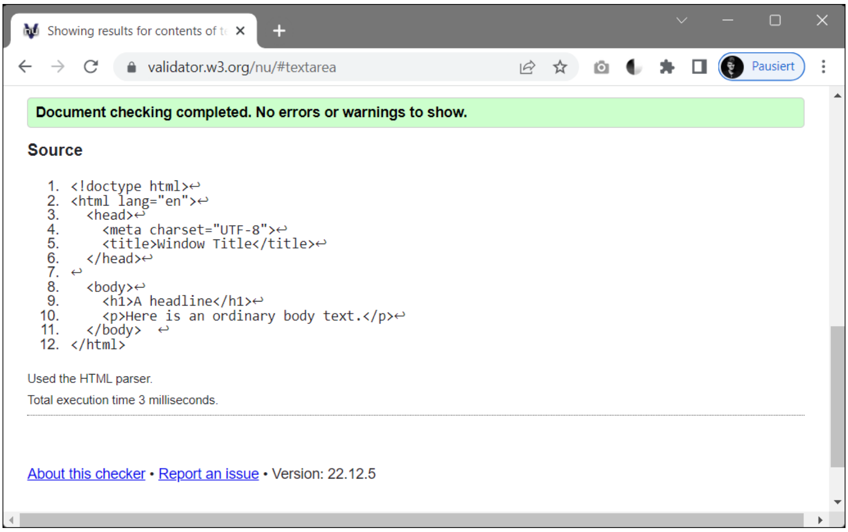
Validate by direct input

Enter the Markup to validate:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Window Title</title>
  </head>

  <body>
    <h1>A headline</h1>
    <p>Here is an ordinary body text.</p>
  </body>
</html>
```

Figure 1.8 HTML Code for Validation Has Been Inserted Here

When you click the **Check** button afterwards, the validation will be performed. If the HTML code was error-free, you'll get a green bar indicating that the HTML document was OK, which is shown in Figure 1.9.


Showing results for contents of t ×
validator.w3.org/nu/#textarea
Pausiert
About this checker
Report an issue

Document checking completed. No errors or warnings to show.

Source

```
<!doctype html>↩
<html lang="en">↩
  <head>↩
    <meta charset="UTF-8">↩
    <title>Window Title</title>↩
  </head>↩
↩
  <body>↩
    <h1>A headline</h1>↩
    <p>Here is an ordinary body text.</p>↩
  </body>  ↩
</html>
```

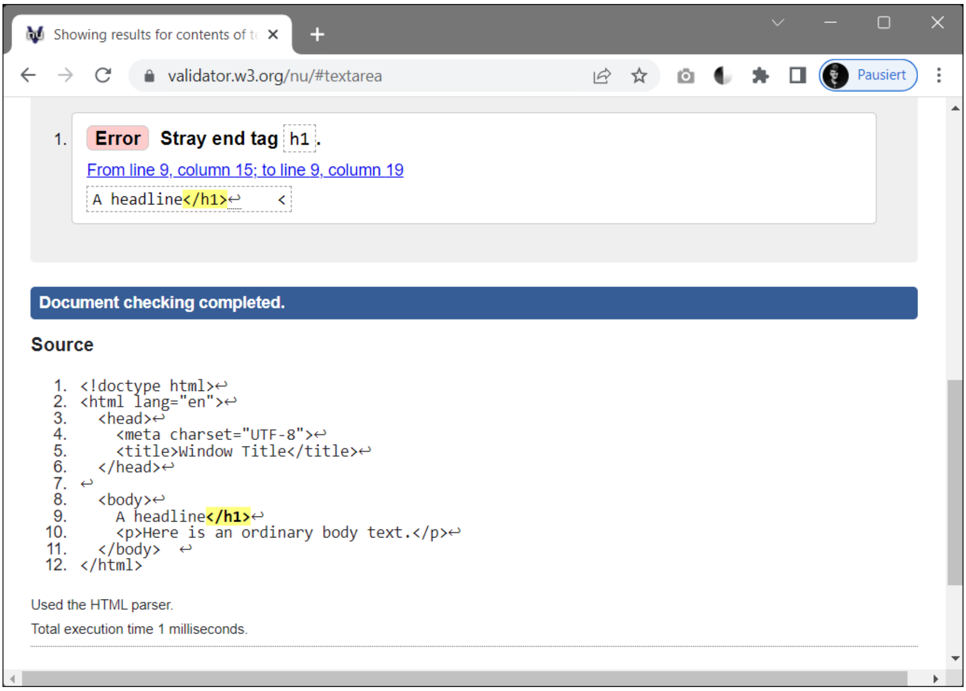
Used the HTML parser.

Total execution time 3 milliseconds.

About this checker • Report an issue • Version: 22.12.5

Figure 1.9 HTML Code Has Passed the Test and Is Valid

If the check was invalid, the error(s) will be listed with a message and marked in the HTML code, as you can see in Figure 1.10. You can read the warnings and error messages if you scroll down a little. Feel free to experiment with your HTML code and intentionally include some typos or simply remove a line in the code. As a beginner, you probably won't be able to do much with the error messages at this point.


Showing results for contents of t
validator.w3.org/nu/#textarea
Pausiert

1. **Error** **Stray end tag** `h1`.

From line 9, column 15; to line 9, column 19

```
A headline</h1>↩     <
```

Document checking completed.

Source

```
1.  <!doctype html>↩
2.  <html lang="en">↩
3.    <head>↩
4.      <meta charset="UTF-8">↩
5.      <title>Window Title</title>↩
6.    </head>↩
7.  ↩
8.    <body>↩
9.      A headline</h1>↩
10.     <p>Here is an ordinary body text.</p>↩
11.   </body>  ↩
12. </html>
```

Used the HTML parser.

Total execution time 1 milliseconds.

Figure 1.10 This Check Resulted in Errors, as You Can See from the Error Message Output

By the way, an error output doesn't mean that the web page can't be displayed. Web browsers are relatively fault-tolerant and also have their own rules. Nevertheless, in the worst-case scenario, the web page *may* not be displayed properly in a particular web browser or—even worse—not at all.

Validation Tip

Enter some web addresses of larger websites under **Validate by URI**. It will probably surprise you that there are hardly any websites with 100% valid code. Some larger websites may well display between 100 and 400 errors. This will surely make you wonder what's the point of writing HTML-compliant code if not even the creators of large websites adhere to it. I'll briefly describe this in the next section.

1.5.5 Good Reasons for Validating the HTML Code Anyway

Validating a web page or HTML code has many advantages as follows:

- **Display in any browser**
  Probably the weightiest argument is the display of the website in any web browser. Errors in the HTML source code can cause the web page not to be displayed or not to be displayed correctly in some browsers. While web browsers are fairly fault-tolerant, especially on mobile devices, less powerful error-correction routines are included in the web browser being used.
- **Search engines**
  The search engines look for text and keywords. What good is the most beautiful website if the search engine can do nothing with the document and the site is therefore not found on the web?
- **Accessible websites**
  People with a physical limitation such as a severe visual impairment are dependent on special preparations of the web offer that go beyond the usual presentation. Faulty web pages with poor or incorrect text markup can cause the read-aloud software, for example, to function incorrectly or incompletely. This is a shortcoming, by the way, of some CMSs, as they often generate code that's less accessible to people with disabilities.
- **Validation**
  Validation is enormously important, especially for beginners, to avoid starting with the wrong things right away. Precisely because HTML is so fault-tolerant, it's easy to be tempted to write messy code. A validation provides initial feedback to beginners.
- **Quality assurance**
  In addition, proper code ensures quality assurance, which means that the website will still work in future browser versions, when they may not be as fault-tolerant.
- **Professionalism**
  Proper code also shows that you're a professional developer who cares about delivering decent work.

There are certainly other reasons for paying attention to clean HTML code. As you'll get to know HTML (and CSS) in this book, you should always keep HTML validation somewhat in mind. Even if you're a budding developer and programmer planning to develop your own web applications or even your own CMS in the future, clean HTML code should always remain your focus. The unfortunate fact is that dynamically created web pages often contain less clean code. The same is true for WYSIWYG editors. Again, the code isn't always validated as clean HTML, but with HTML validation, you can always rework the code manually (if you know how to do that).

More Tools for Validation

You don't have to go to *http://validator.w3.org* every time to validate your HTML code or web page. In this regard, too, there are suitable extensions for every web browser available that can be installed later. For other browsers, you can find *favelets* at *http://validator.w3.org/favelets.html*. Some HTML editors also provide a basic validation of the code. Favelets are small snippets of JavaScript embedded in a bookmark URL that allow bookmarks in browsers to perform various advanced tasks.

1.6 Conventions Used in This Book

The following conventions apply to the examples used in this book: If you find the ellipsis points (...) there, then the code has been shortened for space reasons. The complete and unabridged example, on the other hand, can be found on the website for the book (*www.rheinwerk-computing.com/5695/*) and at *https://html-examples.pronix.de*. The listing caption corresponds to the exact path within the ZIP file. Parts in the listing that have been highlighted in bold are particularly relevant in the example.

```
...
<html>
  <head>
    <meta charset="UTF-8">
    <title>Window Title</title>
  </head>
  <body>
  ...
  </body>
</html>
```

Listing 1.2 /path_to_example/sample_name.html

1.7 Summary

In this chapter, you learned about different types of websites and what's behind terms such as web presence, blog, web store, landing page, and web platform. You now know what dynamic and static websites are. In addition, you've read that HTML, CSS, and JavaScript should be the basic languages of a web developer and that you'll get to know all three in this book. Last but not least, you learned how to create, save, and display an HTML document in a web browser, as well as how to check the HTML code for errors.

Chapter 2

Basic Structure of HTML and HTML Documents

This chapter introduces you to the basic syntax and structure of HTML as a language, as well as the individual components that make up a classic HTML document.

You'll also learn how an HTML document is generally structured. At this point, it's not yet important that you understand the examples and the individual HTML elements. At the end of the chapter, you'll know what HTML tags and HTML elements are and into which sections an HTML document is basically divided, which is sufficient for the time being.

You'll learn about the following important aspects in this chapter:

- How to implement a structure using HTML
- What HTML tags and HTML elements are
- How to correctly nest HTML elements
- What HTML attributes (properties of HTML tags) are
- How to use comments in an HTML document
- How an HTML document is structured
- How to set the document type to `<!doctype>`

2.1 Syntax and Structure of HTML and HTML Documents

This section describes the basic grammar of HTML and the basic structure of HTML documents. You certainly can use HTML without knowing the grammar, but if you really want to learn and use valid HTML, you should know and follow the rules.

2.1.1 How to Structure a Document in HTML

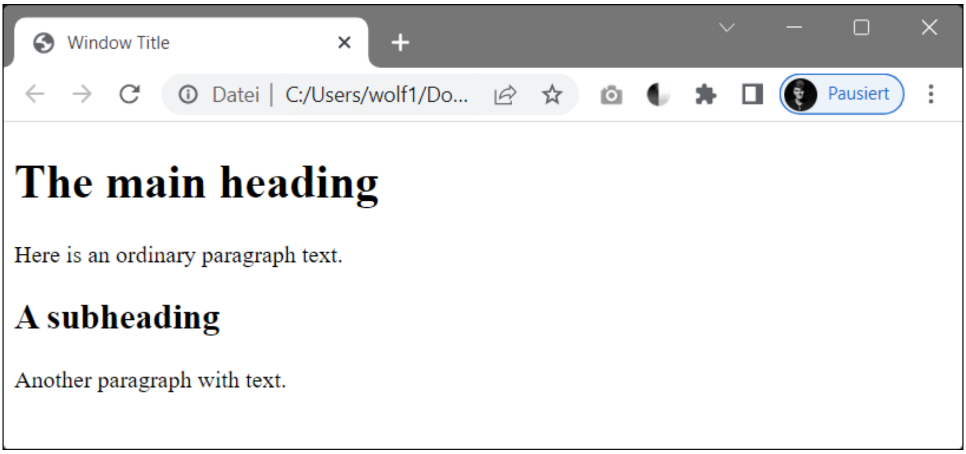
HTML is structured in the same way as you know it from other media or applications. When you look at this book, a newspaper, or even a document in a word processor (e.g., Word), you'll always find some kind of structure. In this book, for example, each chapter contains a heading followed by text with paragraphs and occasionally some pictures. Here and there, you'll also come across some tables. In some sections, subheadings are used at different hierarchical levels. In the same way, an HTML document is structured via HTML elements.

For demonstration purposes, let's take a look at an HTML document with a simple HTML page structure, which will then be explained:

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Window Title</title>
  </head>
  <body>
    <h1>The main heading</h1>
    <p>Here is an ordinary paragraph text.</p>
    <h2>A subheading</h2>
    <p>Another paragraph with text.</p>
  </body>
</html>
```

Listing 2.1 /examples/chapter002/2_1_1/index.html

When you load this HTML document into your web browser, you should see a display similar to Figure 2.1.



The main heading

Here is an ordinary paragraph text.

A subheading

Another paragraph with text.

Figure 2.1 A Structured HTML Document in the Web Browser (Google Chrome)

Figure 2.2 shows the basic elements of the page structure of an HTML document and their meaning. The main focus is on the HTML code and its elements for structured presentation. You'll learn more about the actual basic framework and the individual HTML elements in detail in the course of the next chapters.

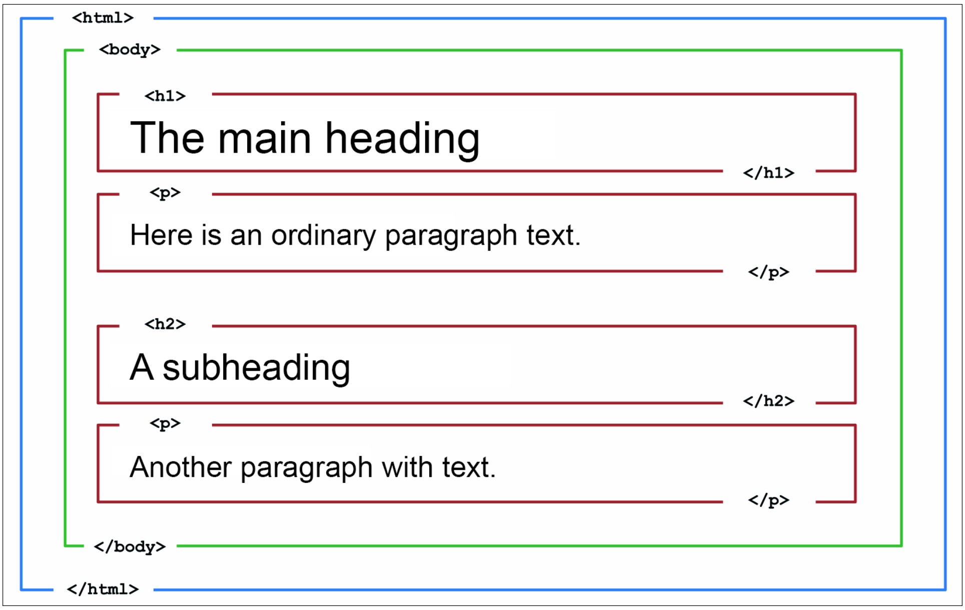


Figure 2.2 Basic Page Structure of an HTML Document

Everything you see here between `<html>` and `</html>` is the HTML code for the HTML document. For this reason, the `html` element is often referred to as the *root element* of an HTML file. There can be only one such root element in an HTML file. This element also summarizes the header data between `<head>` and `</head>`. The part visible in the web browser is written between `<body>` and `</body>`. In this example, you'll find a first-order heading between `<h1>` and `</h1>`, followed by plain paragraph text between `<p>` and `</p>`. This is followed by another second-order heading between `<h2>` and `</h2>`, followed by another paragraph text between `<p>` and `</p>`.

Figure 2.2 also shows you that the individual elements are nested as in a rectangular container, and the HTML document is structured with the HTML elements. Strictly speaking, in this figure, the area between `<html>` and `</html>` should be drawn a bit wider (outside the display area) because it also contains elements that aren't displayed in the web browser.

If you think web pages are assembled from rectangular elements, you're right. Web pages consist of rectangular boxes that are arranged in the browser below each other, next to each other, and inside each other. In Figure 2.3, I made these rectangular boxes visible using CSS. Later in the book, you'll learn how to design such boxes with CSS.

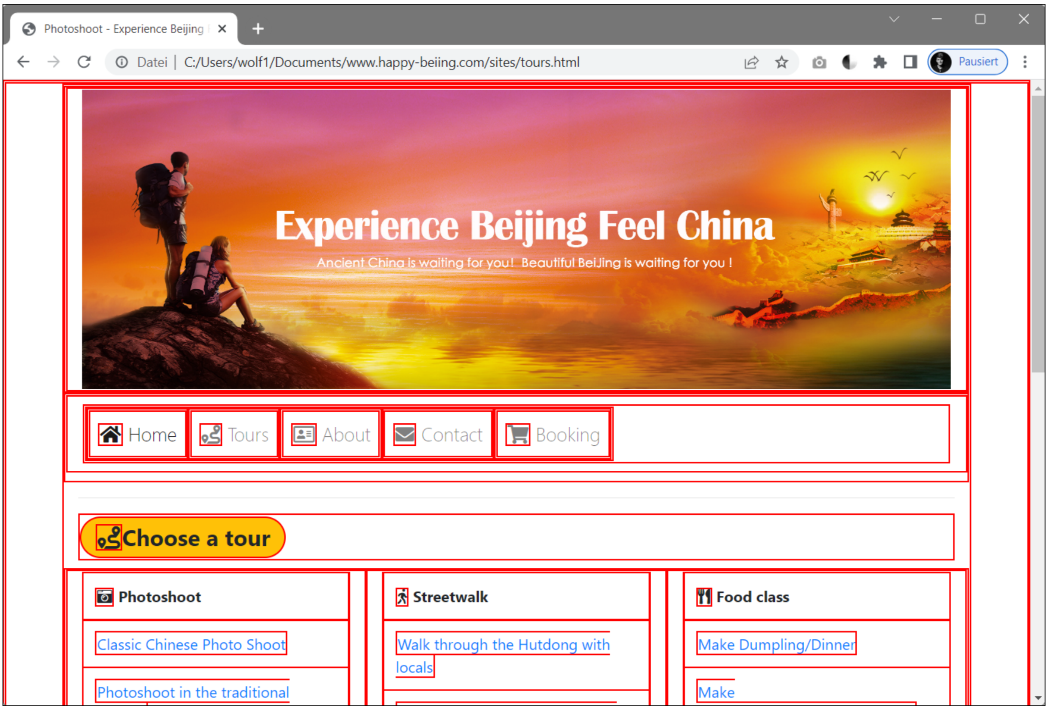

Photoshoot - Experience Beijing
Datei | C:/Users/wolf1/Documents/www.happy-beiing.com/sites/tours.html
Pausiert
Experience Beijing Feel China
Ancient China is waiting for you! Beautiful BeiJing is waiting for you !
Home
Tours
About
Contact
Booking
Choose a tour
Photoshoot
Classic Chinese Photo Shoot
Photoshoot in the traditional
Streetwalk
Walk through the Hutdong with locals
Food class
Make Dumpling/Dinner
Make

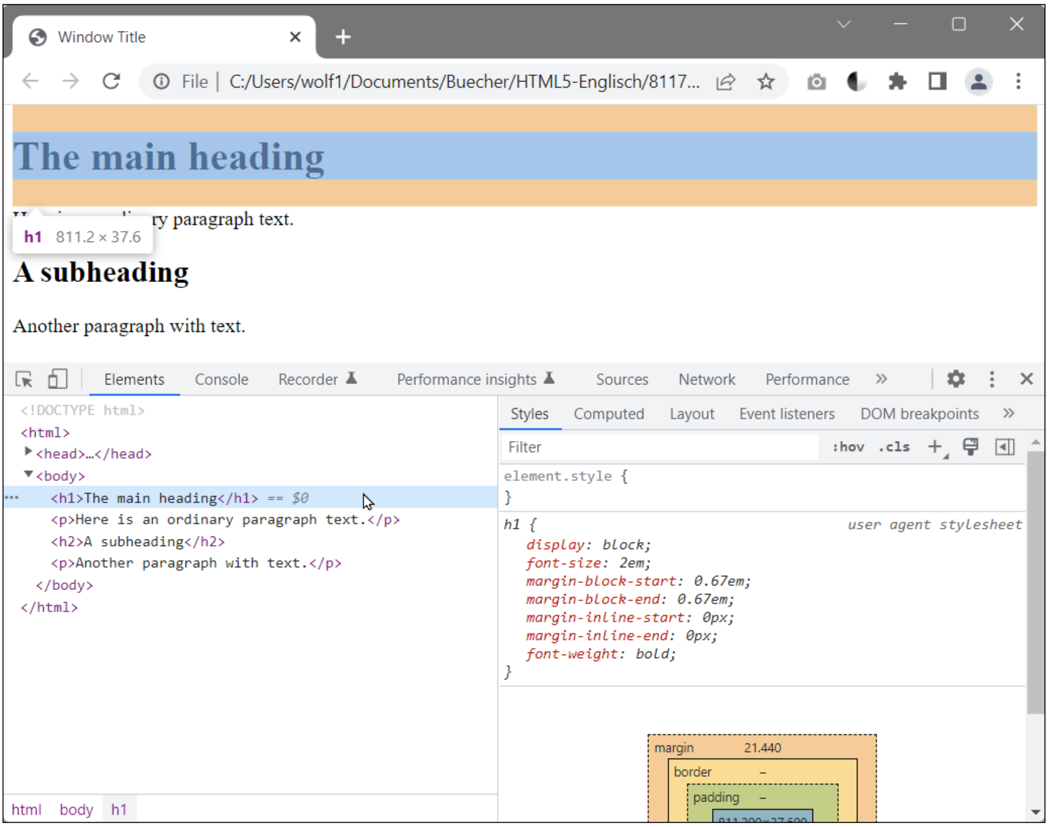
Figure 2.3 The Rectangular Boxes That Make Up a Web Page Have Been Made Visible

2.1.2 Viewing the Tree Structure Using the Document Object Model Inspector

The HTML code of an HTML document consists of pure text. Only a web browser creates a model from this HTML document in the form of a tree structure of objects such as HTML elements, attributes, and text. This model is referred to as the *Document Object Model* (DOM). Each object in this *DOM tree* is referred to as a *node* and can be manipulated via a public interface using JavaScript.

If you want to view or examine this tree structure of HTML elements in your web browser, you can do so with a DOM inspector. All major browser manufacturers provide such web developer tools along with the web browser.

In Figure 2.4, you can see the DOM inspector of the Google Chrome web browser in use. When you look at an example with such a tool, you can clearly see the nested tree structure of HTML. The hierarchical DOM view was called using the DOM Inspector of the Google Chrome web browser (via **More Tools • Developer Tools**).


Window Title
File | C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117...
h1 811.2 × 37.6
ry paragraph text.
Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Styles
Computed
Layout
Event listeners
DOM breakpoints
Filter
:hov
.cls
margin 21.440
border –
padding –
html
body
h1

The main heading

A subheading

Another paragraph with text.

```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
...   <h1>The main heading</h1> == $0
      <p>Here is an ordinary paragraph text.</p>
      <h2>A subheading</h2>
      <p>Another paragraph with text.</p>
    </body>
</html>
```

```
element.style {
}
h1 {                                          user agent stylesheet
   display: block;
   font-size: 2em;
   margin-block-start: 0.67em;
   margin-block-end: 0.67em;
   margin-inline-start: 0px;
   margin-inline-end: 0px;
   font-weight: bold;
}
```

Figure 2.4 Hierarchical DOM View

2.1.3 HTML Tags and HTML Elements

In the previous section, you saw how different HTML elements such as `<h1>...</h1>`, `<h2>...</h2>`, and `<p>...</p>` were used to describe the page structure. A complete *HTML element* usually consists of an opening HTML tag, a closing HTML tag, and everything in between. For example, the following line represents a complete HTML element:

```
<tagname>Text within the HTML element</tagname>
```

Instead of `tagname`, real HTML keywords describing different parts of a web page are used for this purpose. For example, you can represent a first-order heading using the following line:

```
<h1>HTML element as heading</h1>
```

An HTML element is usually the completely displayable element such as a heading, paragraph text, or an entire hyperlink. The HTML elements, in turn, are marked by *HTML tags*.

The HTML tags (also called *HTML markup tags*) are keywords surrounded by angle brackets, such as `<p>`. Most HTML tags come as a pair, such as `<p>` and `</p>`. The first tag of the pair is the start tag, and the second one is the end tag. In practice, these tags are also called *opening tag* (= start tag) and *closing tag* (= end tag). Both tags have the same tag name, except that the closing tag is terminated by a *forward slash* (e.g., `</p>`).

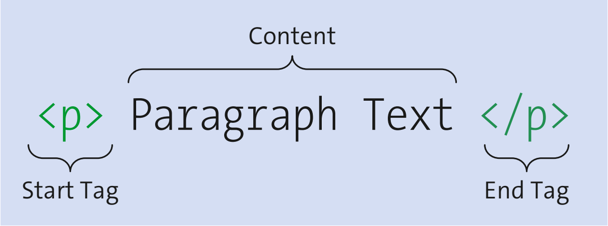


Figure 2.5 A Complete HTML Element with Its Individual Components (Start Tag, Element Content, and End Tag)

2.1.4 Nesting HTML Elements and the Hierarchical Structure

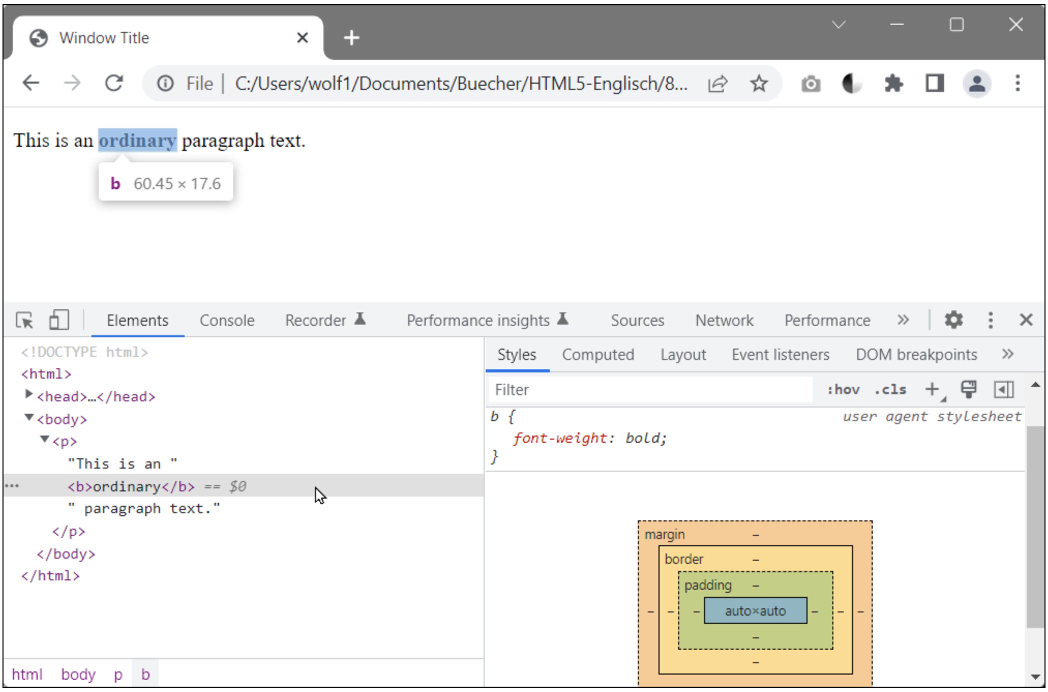
Most HTML elements can be nested and contain other HTML elements. Such nesting creates a hierarchical structure. The following example demonstrates such a simple nesting of HTML elements:

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Window Title</title>
  </head>
  <body>
    <p>This is an <b>ordinary</b> paragraph text.</p>
  </body>
</html>
```

Listing 2.2 /examples/chapter002/2_1_4/index.html

Here, another HTML element has been nested within the paragraph text between `<p>` and `</p>`. The `b` element makes sure that the text is displayed in bold font (`b` = *bold*). In the example, the HTML element from `<b>` to `</b>` is actually the child element of the HTML element from `<p>` to `</p>`. Strictly speaking, the HTML element from `<p>` to `</p>` is again just a child element of the HTML element from `<body>` to `</body>`. This creates a fairly structured markup. Complex HTML documents therefore often contain deep nesting.

When you take a look at the DOM inspector in Figure 2.6, you'll see the structured markup from the parent `<html>` element, through the child `<body>` element, and the child-child `<p>` element, to the innermost HTML element, the child-child `<b>` element.


Window Title
File | C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8...
b 60.45 × 17.6
Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Styles
Computed
Layout
Event listeners
DOM breakpoints
Filter
:hov
.cls
user agent stylesheet
margin
border
padding
auto×auto
html
body
p
b

This is an ordinary paragraph text.

```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<p>
      "This is an "
      <b>ordinary</b> == $0
      " paragraph text."
    </p>
  </body>
</html>
```

```
b {
   font-weight: bold;
}
```

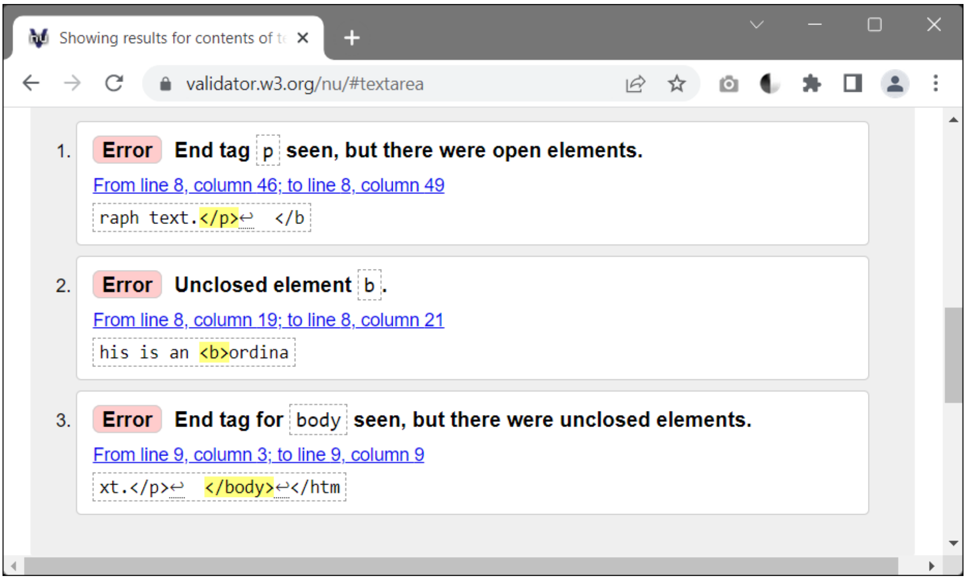
Figure 2.6 A DOM Inspector Lists the Hierarchical Structure Very Clearly

2.1.5 Avoiding Incorrect Nesting of HTML Elements

It's important to always ensure that a child element is completely contained within the parent element. This means that you have to write an end tag of a child element within the parent element, that is, before the end tag of the parent element. In this context, you should take a look at the following erroneous example:

```
...
<body>
  <p>This is a <b>common paragraph text.</p></b>
</body>
...
```

When you validate this HTML code, three error messages get returned at once, as you can see in Figure 2.7. First, it's noted that an end tag `</p>` was used, although it still contains open elements (here, only `<b>`). Then the start tag `<b>` is also noted as not having been closed. Finally, the end tag `</b>` is described as invalid at the end because no start tag `<b>` was found in the corresponding scope here.


Showing results for contents of t
validator.w3.org/nu/#textarea

1. Error End tag `p` seen, but there were open elements.
   From line 8, column 46; to line 8, column 49
   `raph text.</p>↩ </b`
2. Error Unclosed element `b`.
   From line 8, column 19; to line 8, column 21
   `his is an <b>ordina`
3. Error End tag for `body` seen, but there were unclosed elements.
   From line 9, column 3; to line 9, column 9
   `xt.</p>↩ </body>↩</htm`

Figure 2.7 Incorrect Nesting Is Immediately Detected by Means of Validation

You can fix the error by writing the end tag `</b>` into the section, which in this case is between `<p>` and `</p>`, because the start tag `<b>` is also in it. You must always close the innermost elements first and only then the outer ones. The correct notation of the preceding example looks as follows:

```
...
<body>
  <p>This is a <b>common paragraph text.</b></p>
</body>
...
```

2.1.6 Omitting the End Tag of an HTML Element

In HTML, there are some optional tags—that is, HTML tags—that you could omit. For example, under certain conditions, you can omit the end tag, as shown in the following code snippet:

```
...
  <body>
    <p>This is an ordinary paragraph text.
    <p>This is another paragraph text.
  </body>
...
```

Here, the closing end tags `</p>` have been omitted, which works and is permitted in most web browsers. In addition, it's possible to omit the `<html>` start tag and `</html>`

end tag or even `<body>` and `</body>` under certain conditions. Missing tags are inserted in the right places by the web browser when the DOM tree gets generated. When you use the DOM inspector and look at an HTML element with a missing end tag and the same one without a missing end tag, you'll notice that the web browser always ends up generating the same HTML code from the document.

However, omitting tags requires that you know and follow the rules defined for doing so. For example, it isn't possible to omit an end tag just like that for all HTML elements. Many HTML elements produce an unexpected result or error when the end tag is missing. You can, for instance, omit the end tag for `p`, `ul`, or `li` elements, but not for `div` elements. These somewhat inconsistent requirements make it harder rather than being helpful, especially for beginners. For this reason, theoretically and practically, you can write the example */examples/chapter002/2_1_4/index.html* without the `html` and `body` tags as follows, and still the web browser would create the same DOM tree from it as shown earlier in Figure 2.6.

```
<!doctype html>
<meta charset="UTF-8">
<title>Window Title</title>
<p>This is a <b>common</b> paragraph text.</p>
```

Listing 2.3 /examples/chapter002/2_1_6/index.html

More Information Online

For an overview of the circumstances under which you can omit specific tags, see *https://html.spec.whatwg.org/multipage/syntax.html#optional-tags*.

2.1.7 Standalone HTML Tags without End Tags

Some HTML elements are *standalone tags* (or *void tags*) that have no content and therefore don't require an end tag. An example of such a tag without contents is `<br>`, which causes a line break:

```
<p>A line break<br>The next line</p>
```

However, the HTML element `<br>` shouldn't be misused to increase the spacing between two lines. For this purpose, we can use either CSS (e.g., with the `margin` feature) or the `p` element.

Case Sensitivity of HTML Tags

In HTML, you can capitalize and lowercase the names of the tags as they aren't case-sensitive: `<h1>` and `<H1>` mean the same thing. We use lowercase throughout this book.

2.1.8 Additional HTML Attributes for HTML Elements

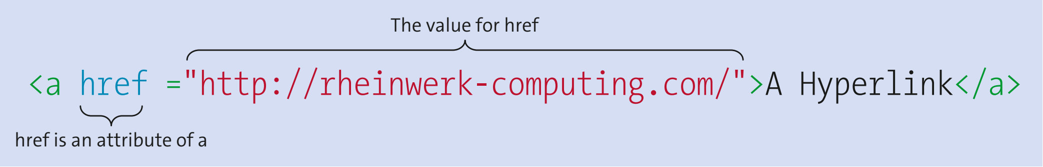
The HTML elements can contain additional *attributes* (sometimes also referred to as *properties*) that you can use to specify additional information about an element. You can use attributes only for the start tags and the standalone tags. The attribute details are specified in an attribute name-value manner such as `name="value"`. Let's take a look at some simple examples:

```
...
<body>
  <p lang="en">
    Please <a href="http://rheinwerk-computing.com/">click here</a>.
  </p>
  <p>
    <img src="cover.png" alt="Book cover">
  </p>
</body>
...
```

Listing 2.4 /examples/chapter002/2_1_8/index.html

You can use the `lang="en"` attribute to specify the language used in the `p` element. `lang` stands for *language* and `en` for *English*. The `a` element allows you to define a hyperlink to another page. Without the `href` attribute, the HTML element wouldn't make any sense at all here as it specifies the URL of the page (here, *http://rheinwerk-computing.com/*) to which the link should go when the user clicks on the text written between `<a>` and `</a>`. The same applies to the standalone `img` element, where you specify the URL to an image (here, *cover.png*) via the `src` attribute. In addition, for an `img` element, you must specify the `alt` attribute for an alternative text (here: `Book cover`).

The `img` element shows that you can use more than one attribute for HTML elements. The order in which you note the attributes in the HTML element is arbitrary. For example, for the `img` element, you could specify the `alt` attribute first and then the `src` attribute. When you use multiple attributes, there must be at least one space between an attribute name-value pair. It's recommended to write the values of attributes between quotation marks, for example, `"value"`. Most of the time, double quotes are used for this, although single quotes (e.g., `'value'`) are also permitted. The reason for this recommendation is the downward compatibility.



```
<a href ="http://rheinwerk-computing.com/">A Hyperlink</a>
```

Figure 2.8 HTML Elements Can Contain Additional Attributes

It’s probably unnecessary to mention that certain attributes can only be used for certain elements. For example, you can’t use the `href` attribute in a `p` element. Nevertheless, there are also global attributes in HTML, such as `lang`, which you can use in almost all HTML elements.

2.1.9 Using Comments in HTML Documents

If you want to comment on an HTML code at any place, you can introduce such a comment via the `<!--` string and conclude it with the `-->` string. Everything you write between `<!--` and `-->` will be suppressed by the web browser and won’t be displayed. Here’s a simple example with comments:

```
...
<body>
  <!-- Find a meaningful header -->
  <h1>Header</h1>
  <!--
       Think about what fits
       into the paragraph for this heading.
  -->
  <p>A lot of text</p>
  <!-- <p>A second paragraph with text</p> -->
</body>
...
```

The lines in bold are comments, which are suppressed and won’t be displayed by the web browser. As you can see, this also applies to HTML elements such as the `p` element in the example. Here, the complete `p` element has been commented out so that it gets ignored by the web browser for rendering.

Warning: Comments Can Be Viewed in the Source Code

Comments are quite useful to add annotations at certain places in the HTML code or information about the creation date or other *credits*. However, even if the comments aren’t displayed in the web browser, they remain in the source code. This means you should always think twice about what kind of comments you write because every visitor can see the source code.

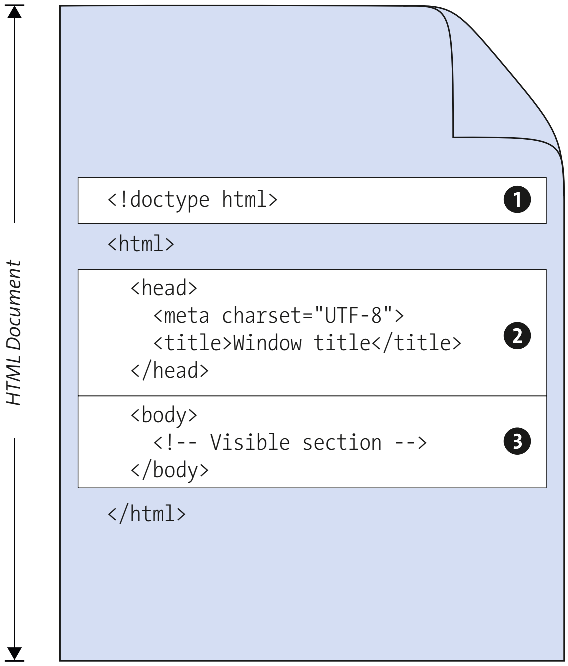
2.2 A Simple HTML Document Framework

The basic framework of an HTML document roughly consists of three parts, as you can see in Figure 2.9:

❶ The *HTML document type* specifies the HTML version used.

❷ The *header area* usually contains the nondisplayable information about the document.

❸ The *document body* contains the displayable content for the web browser.



```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Window title</title>
  </head>
  <body>
    <!-- Visible section -->
  </body>
</html>
```

Figure 2.9 The Subdivision of an HTML Document

2.2.1 HTML Document Type: <!doctype>

The `<!doctype>` declaration must be the first specification in an HTML document, before the `<html>` tag. The `<!doctype>` isn't an HTML tag, but an instruction for the web browser about the HTML version in which the web page was created.

In the old HTML 4.01 or XHTML 1.0, this `<!doctype>` declaration still required a *document type definition* (DTD) based on standard generalized markup language (SGML). This DTD specified the rules for the markup language so that web browsers could correctly render the content according to the DTD.

The current HTML is no longer based on SGML, and thus no `<!doctype>` declaration would be needed at all, so you can write the following here:

```
<!doctype html>
```

This line is used by the web browsers that require the presence of a `<!doctype>` declaration. That version is understood by all web browsers, even those that don't know the current HTML at all. As a result, this **`<!doctype html>`** is only used to ensure downward compatibility with older web browsers. By the way, the `<!doctype>` declaration isn't case-sensitive, and you could also use `<!DOCTYPE html>`.

2.2.2 Beginning and Ending an HTML Document: <html>

After `<!doctype html>` follows the root element, `html`, which informs the web browser that the page has been written in HTML code. The root element encloses all other elements between the `<html>` start tag and the `</html>` end tag—you could also say the `html` element is the container for all other HTML elements.

Even if you don't need to, in practice, you can declare the attribute of the website's language right away (e.g., `lang="en"` for English). Users who use a screen reader will be grateful to you.

Specifying the Language

The specification of the language via the HTML attribute `lang` is a global attribute and specifies the content language of the element. Thus, the attribute isn't limited to `html` and can be used in almost all HTML elements. The speech markup helps screen readers use the correct speech output and helps search engines match the content. The web browser can use this specification to correctly display typical special characters of a language, for example. Such speech codes can consist of two parts. In addition to the primary language code, you can specify an optional subcode. For example, via `lang="en-UK"`, you can use the UK version of English.

As direct child elements of the `html` element, only the elements `head` and `body` are allowed.

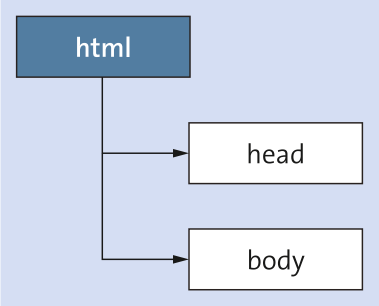


Figure 2.10 Below <html>, You'll Find <head> and <body>

2.2.3 Head of an HTML Document: <head>

The head area between `<head>` and `</head>` defines various things that, with the exception of the `title` element, aren't used directly for display in the web browser. In that area, you can specify information that gets evaluated by the web browser and search engines. This can involve the insertion of scripts, instructions for the web browser on where to find a stylesheet, and various kinds of metadata with information about the HTML document itself. I'll go into more detail about the individual elements that can be used in the head area between `<head>` and `</head>` of an HTML document in Chapter 3.

2.2.4 Visible Part of an HTML Document: <body>

The displayable document body is specified in HTML between `<body>` and `</body>`. Everything in between those two tags—such as text, hyperlinks, images, and tables—gets displayed in the web browser. Thus, unlike the `head` element, the `body` element is the displayable area of an HTML document.

2.3 Summary

This chapter was of a more theoretical nature, but it's imperative that you know about the basic structure of HTML and an HTML document. Following are the most important aspects of this chapter that you should definitely understand:

- HTML tags and HTML elements
- HTML attributes
- Correct nesting of HTML elements
- Document type `<!doctype html>`
- Head area and displayable area basic sections of an HTML document

Chapter 3

Head Data of an HTML Document

The head data between `<head>` *and* `</head>` *contains important information and data about an HTML document that's used by web browsers or search engines. In this chapter, you'll get to know the HTML elements for the head data of an HTML document in more detail.*

In the head of the HTML document between `<head>` and `</head>`, you can insert various HTML elements that enable you to control the content and the display of a web page. You can also establish the relationships between the web browser and other pages or documents here. The content you write in the `head` element doesn't get displayed by the web browser, except for the `title` element. An overview of the different HTML elements, which you can write in the head section between `<head>` and `</head>`, is followed by the description of the individual HTML elements.

This chapter may not be spectacular and exciting, but even the nonvisible parts in the head of an HTML document are part of the essential basics of HTML. If you're in a rush, I recommend that you at least take a look at the sections on the `<meta>` elements (Section 3.8.1, specifically the character encoding) and `<title>` (Section 3.2). For the time being, these two headers are the most important elements for the next chapters. For all other HTML elements for head data, you can always look them up here if needed.

3.1 Overview of HTML Elements for the Head

You must write the `head` element with the head data of an HTML document directly after the opening `<html>` tag and before the document body with the `body` element. You can use the elements from Table 3.1 or Figure 3.1 between `<head>` and `</head>` (the order doesn't matter). Of the elements listed in Table 3.1, you must specify at least the `title` element.

HTML Element	Meaning
`<title>...</title>`	Contains the title of the HTML document.
`<base>`	Sets base URLs/targets for all relative URLs in a web page.

Table 3.1 Elements That Can Be Used in the HTML Document Head between <head> and </head>

HTML Element	Meaning
`<link>`	Sets logical links of the HTML document to other files to be included.
`<style>`	Sets the local stylesheet rules for the HTML document.
`<script>`	Integrates the client-side scripts. The `script` element isn't restricted to the HTML document header and may also appear (multiple times) in the document body.
`<meta>`	Sets the metadata such as keywords, descriptions, or the character set for the HTML document.

Table 3.1 Elements That Can Be Used in the HTML Document Head between <head> and </head> (Cont.)

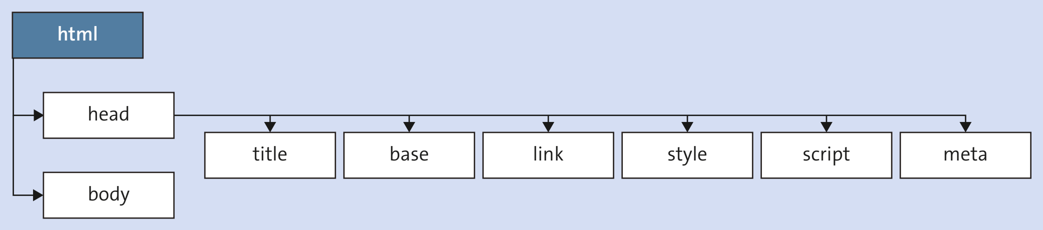


Figure 3.1 In the Head Element between the <head> and </head> Tags, You Can Use the <title>, <base>, <link>, <style>, <script>, and <meta> Elements

3.2 <title>: Heading of the HTML Page

In every HTML document, you should use a title that gets displayed in the header of the web browser. You can write such a title between the `<title>` and `</title>` tags within the `head` element. If no title is used, what appears here depends on the web browser. Often, you'll then find a title such as **Untitled Document** or **Untitled**, for example. You can use only one `title` element in total for an HTML document.

In Figure 3.2, you can see the `title` element used in the following example being rendered:

```
<!doctype html>
<html lang="en">
  <head>
    <title>–The Heading of the HTML Page</title>
    <meta charset="UTF-8">
  </head>
...
```

Listing 3.1 /examples/chapter003/3_2/index.html

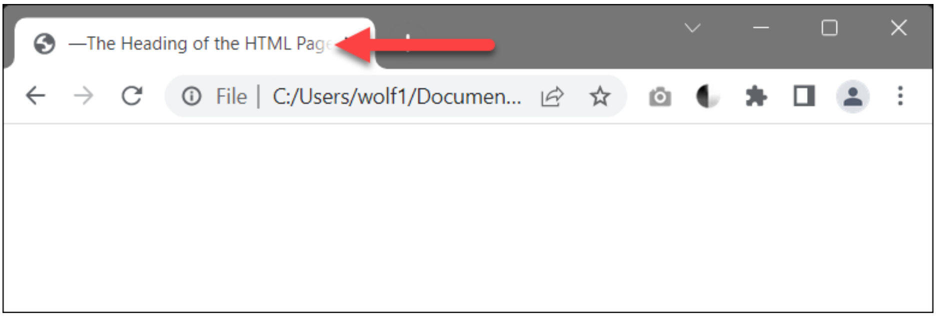
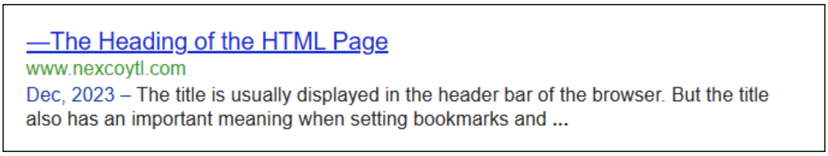

—The Heading of the HTML Pag
File | C:/Users/wolf1/Documen...

Figure 3.2 The Title Is Usually Displayed in the Header Bar and/or Tab of the Web Browser

Invalid HTML

If you omit the `title` element, an error message displays when you check the validity of the HTML code (e.g., at *http://validator.w3.org*). This means a `title` element must exist in every HTML document for it to be valid HTML. In addition, no other HTML elements are allowed in between `<title>` and `</title>`.

Besides the display in the header bar or tabs of the web browser, there are two much more important reasons to use the `title` element: first, this title is used as a name suggestion by the web browser when setting bookmarks (favorites), and second—and even more importantly—the title has a high significance with the search engines on the web. As you can see in Figure 3.3, the title is often the first thing your visitors see when your website is listed in a search engine. You can create such a search result yourself for testing purposes with a *SERP snippet generator* (e.g., *www.serpsimulator.com*). It often occurs that the title is also a clickable link to your website. In addition, the title is important for the hit list and for the rank of the page in the hit index of the search engine.



—The Heading of the HTML Page

www.nexcoytl.com

Dec, 2023 – The title is usually displayed in the header bar of the browser. But the title also has an important meaning when setting bookmarks and ...

Figure 3.3 For Search Engines, the Importance of the <title> Element Shouldn't Be Ignored

Titles for Search Engines

The topic of *search engine optimization* (SEO) can't be described in a few paragraphs and encompasses many subareas. It isn't possible either to make a general statement about when exactly something is "optimal." Even the SEO experts often disagree on this. Nevertheless, it's safe to say that the title is very important in search engines. It's often recommended to use one or two keywords, followed by a slogan and perhaps the web address (e.g., *Keyword 1, Keyword 2—A short heading—www.domain.com*). In practice, for example, the following title would make sense: *Smartphones, Cell Phones—buy cheap smartphones—www.domain.com*. In addition, the title shouldn't be too long, otherwise it will be truncated when listed in the search engine. The common recommendation ranges between 60 and 70 characters.

3.3 Related Topic: Naming Convention and Referencing

At this point, it's necessary to go into the naming conventions for files, directories, and directory structures when referencing other content because you'll make use of them repeatedly in the following sections and throughout this book. Mind you, this isn't yet about the HTML elements or HTML tags you can use to link a web page, but only about how to write such a link to the target. If you're already familiar with terms such as full URL or absolute or relative path, you can skip this section or just skim through it.

3.3.1 Valid and Good File Names for an HTML Document

The use of file names for web pages has become quite flexible. Nevertheless, here are some guidelines and recommendations you can follow: Use only lowercase letters a–z, digits 0–9, hyphens, and underscores if possible. The dot is usually used only to separate the file extension. Whether you use uppercase letters in the file name is a matter of taste, but you should keep in mind that some systems are case sensitive while others aren't.

Good Names for the Search Engine

The file names can also be used to place keywords in them for the search engines. Instead of using the relatively meaningless *domain.com/page01.html*, you should choose a better name such as *domain.com/smartphones.html*. If there are several keywords in a file name, you should separate them with a hyphen. The underscore as a separator, on the other hand, is usually not evaluated by search engines as a word separator. Instead of *domain.com/smartphonesandcellphones.html*, where the keywords aren't recognized, it's better to use *domain.com/smartphones-and-cellphones.html*.

3.3.2 Valid Directory Names and Meaningful Directory Structures

The same thing I just wrote for file names applies to the use of directory names. In addition to a meaningful directory name, a meaningful directory structure is also important. Again, you may have the advantage that this structure with good directory names will have a positive effect on your page ranking with the search engines. For example, the following is a useful directory structure:

```
/smartphones
/smartphones/apple
/smartphones/android
/apps/apple
/apps/android
...
```

Thanks to a directory structure based on different topics and coupled with a good file name, you could, for example, call an HTML document named *buy-ramsung-xyz.html* as follows:

```
domain.com/smartphones/android/buy-ramsung-xyz.html
```

On the other hand, you should refrain from using directory names that don't carry much meaning for visitors or search engines, such as the following:

```
/html
/html/pages
/contents/
/contents/pages
...
```

3.3.3 Writing a Reference to a Data Source

Without the functionality to reference other content, the internet wouldn't be what it is today. In addition to classic hyperlinks to other content, such referencing is also used for many other things, including images, external scripts, CSS files, or video resources. For this reason, this section describes several ways to create a reference to other content.

Simple Structure of Addresses on the Internet (URL)

There's no comprehensive treatise at this point, but you should at least know the basic form of an address on the internet, also referred to as a *Uniform Resource Locator* (URL). Only thanks to this URL is it possible to use an address on the internet in a readable format and to access directories and documents. A classic URL looks like this: *http://www.domain.com/path/file.html*.

If you decompose this address, you'll get the individual components listed in Table 3.2.

Protocol	Host Name	Path	File
http://	*www.domain.com*	*path*	*file.html*

Table 3.2 Rough Structure of an Internet Address

You can use the *protocol* (or *scheme*) to specify how the resource should be used. *http://* is the protocol for hypertext documents (*HTTP*). Other well-known representatives are *https://* for a secured data transfer, *ftp://* for a file transfer (*FTP*), or *file://* for the access to local files. With *www.domain.com*, you have a *host name* that's converted to an IP address via the Domain Name System (DNS). It's followed by the *path specification*, whose components are separated from each other by a */*. Finally, the *file name* (here, *file.html*) of the document you want to call is often specified as well.

The *www* of *www.domain.com* is part of the host name and conveniently chosen to give an indication of the host's intended use, for example, as a WWW or FTP server. However, this isn't a prerequisite. The name of a WWW server doesn't necessarily have to start with *www*. You can select the service running on the server with the protocol and port on which the service is listening, for example, port 80 for WWW or port 21 for FTP.

In this example, with the host name *www.domain.com*, it's a WWW domain name for a network server, where the ISO country code *com* is the top-level domain. The top-level domain is sorted either thematically (e.g., *com*, *org*, *net*) or geographically by country (e.g., *de*, *at*, *ch*). *domain* is the second-level domain and the actual name of the server. Here, further subdomains (or sublevels) are possible, which lie below another one in the hierarchy. *example.domain.com* would thus be a subdomain of *domain.com*, for example.

Homepage: "index.html"

If you enter an internet address such as *http://www.domain.com/* in the address field of the web browser, you'll still get a web page displayed in the browser even though you haven't explicitly specified a path or file name there. This is because the web servers return a default page, depending on the setting. Many web servers return at least *index.htm*, *index.html*, or *default.html* if there's a corresponding file in the root directory. This usually works with any other directory as well. For example, if you enter *http://www.domain.com/travel/* in the browser's address bar and there's an *index.html* file in the */travel* directory, that file will be returned. However, as already mentioned, it depends on the settings of the web server what can be returned or done and how this happens. Some web hosts also allow you to define your own rules for this in *.htaccess*. The *.htaccess* file is a configuration file that can be used to make various settings and specifications about things such as access control, exclusion of addresses, error messages, password protection, and alternative content, among other things.

Using a Reference with Full URL to the Data Source

When something is referenced with a full URL, we're talking about the fully spelled out web address. You can use a full URL such as *http://www.domain.com/travel/index.html* or *http://www.domain.com/pictures/foto.jpg* if the data is located on a different machine (host name/domain) than the HTML document. You already learned how a complete URL looks in detail in the previous section.

A Reference as an Absolute Path Specification Relative to the Base URL

If you reference something with an absolute path, the desired data is on the same computer as the HTML document. If a web page is accessible via *http://www.domain.com/travel/index.html*, */travel/index.html* represents the absolute path specification relative to the URL *http://www.domain.com*. Thus, you can use this path specification for your web pages if the data is located within your domain (or subdomain).

Root Directory

The root directory / (the highest directory to reach) of a domain such as *http://www.domain.com* is often set as the document start directory by the web server when the domain is configured. For example, if you connect to an FTP client and want to upload your web pages, this root directory can also be inside a directory named *www*, *htdocs*, *web*, and so on. Nevertheless, the root directory for *http://www.domain.com* is usually still / and not */www*, */htdocs*, or */web*. However, this again depends on the configuration implemented by your web hosting provider, whom you should contact in case of doubt.

Specifying a Reference with a Relative Path to the Data Source

You can use a relative path specification if you use the current address as the reference address. For example, if you're at the full URL *http://www.domain.com/travel/index.html*, and there's an image named *photo.jpg* in the */travel* directory, you can reference this file with a relative path specification such as *photo.jpg* or *./photo.jpg*. Alternatively, you could use the absolute path specification with */travel/photo.jpg* or the absolute URL with *http://www.domain.com/travel/photo.jpg*. The use of the absolute URL is uncommon in such cases.

If *photo.jpg* or *./photo.jpg* is specified as a relative path, it's assumed that the file is located in the same directory. If you want to reference a file in a directory one level above, you can use ../. For example, if you want to access *photo.jpg* in the */travel* directory from the full URL *http://www.domain.com/travel/california/index.html* as a relative path, you could do so by using *../photo.jpg*. This way, you reference the directory above the current directory.

3.4 Defining the Base URL of a Web Page Using <base>

The `base` element allows you to define a base URL or destination for all files referenced in the HTML document. By defining such a base URL, you can use a relative or absolute address to the file in the document as if this file were located on the same host or computer as the HTML document.

It sounds more complicated than it actually is. For this reason, let's take a look at the following simple example that demonstrates the `base` element in practice:

```
<!doctype html>
<html lang="en">
  <head>
    <title>Defining a Base URL</title>
    <base href="https://static.sap-press.com/img/"
          target="_blank">
    <meta charset="UTF-8">
  </head>
  <body>
    <img src="rheinwerk-sappress-logo-header.svg" alt="Logo">
  </body>
</html>
```

Listing 3.2 /examples/chapter003/3_4/index.html

By specifying `href="https://static.sap-press.com/img/"`, the web browser will replace all URLs that weren't fully referenced with `https://` with the base URL *https://static.sap-press.com/img/*. In this example, the image source of `src` (here with `rheinwerk-sappress-logo-header.svg`) is therefore supplemented by *https://static.sap-press.com/img/rheinwerk-sappress-logo-header.svg* in the line that contains the `img` element, so that the Rheinwerk Publishing logo gets displayed in the web browser, which you can see in Figure 3.4.

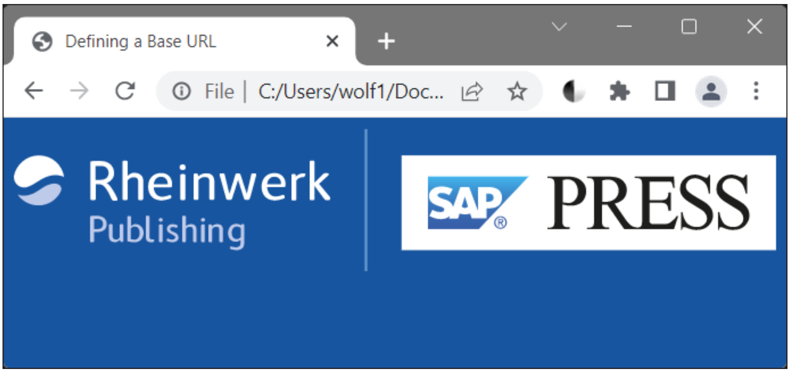


Figure 3.4 Thanks to the Base URL Defined in <base> in the "href" Attribute, the Image File That's Not Fully Referenced Is Supplemented by the Base URL of the Browser and Displayed

Internet Connection Required for Local Testing

For the example shown in Figure 3.4 to work on the local computer, you need a connection to the internet because when the file (here, *rheinwerk-sappress-logo-header.svg*) is called, an attempt is always made to fetch the file from the base URL (here, *https://static.sap-press.com/img/*). Thus, when you use the `base` element, it isn't possible to test a website offline on a local computer. If there's no connection to the internet in the example just shown, only the alternative text of the `alt` attribute gets displayed in the `img` element (here, `"Logo"`).

The `target` attribute, on the other hand, enables you to specify the target where each reference should be opened and displayed. The `_blank` value allows you to make sure in this example that a new window or tab is addressed. In our example, this has no effect because only one image is displayed. For other possible values for the `target` attribute, see Table 3.3.

There can be only one `base` element in an HTML document, and it must be written between `<head>` and `</head>`. If you define multiple `base` elements nevertheless, the web browser will usually use only the first `href` and the first `target` attribute. All the others will be ignored. However, the HTML validity check would return an error if more than one `base` element was used. Furthermore, you must define the `href`, the `target` attribute, or both in the `base` element.

In old HTML 4.01, the attribute value of `target` was a name or keyword for a frame. In current HTML, it's now the keyword for a *browsing context*, which can be a browser window, a browser tab, or even an inline frame (`iframe`).

Attribute	Description
`href`	Defines the base URL. This URL is used by the web browser as the base address for relative or absolute path specifications in the document and is supplemented with this base URL.
`target`	Specifies the target window in which the link target should be displayed. Possible values and their meaning are as follows: ■ `_self`: Opens the reference in the current window. This is the default setting if `target` hasn't been used. ■ `_blank`: Opens the reference in a new window or tab. ■ `_parent`: Opens the reference in the parent window. The parent window is the window from which the current window was opened. If there's no parent window, this option behaves like `_self`. ■ `_top`: Loads the reference of the file in the window that's highest in the hierarchy. If there's no higher parent window at all, this option behaves like `_self`.

Table 3.3 Attributes for the <base> Element

Frames and Framesets

A *frame* is a section of an HTML page into which another HTML page can be included. The combination of multiple frames used to be referred to as a *frameset*. Framesets with the old HTML element `<frameset>…</frameset>` are no longer supported in the current HTML and are obsolete. Alternatives include inline frames with the `iframe` element, CSS, or other server-side techniques.

3.5 Referencing an External Document via <link>

The `link` element is a standalone tag for the head of the HTML document between `<head>` and `</head>`, which you can use to create a relationship between the current document and an external document. In practice, the `link` element is often used to include an external CSS file in the current document. The `<link>` tag can be used more than once in the `head` element to include multiple resources in the current document. The type or purpose of the relationship depends on the value of the `rel` attribute used (see Table 3.4 for the attributes of the `link` element).

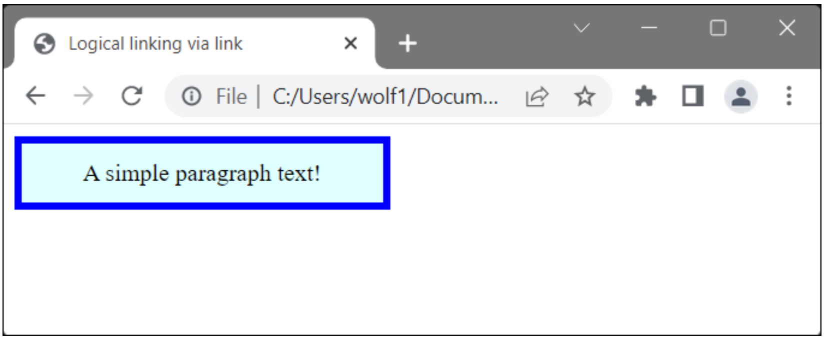
Even though the tag here is `<link>`, this standalone element has nothing in common with the familiar hypertext links, which are underlined and allow the user to navigate to other websites by clicking on this underlined text.

The following example shows you how you can include an external CSS file in the HTML document by using the `link` element.

```
<!doctype html>
<html lang="en">
  <head>
   <title>Logical linking via link</title>
   <link rel="stylesheet" type="text/css" href="style.css">
   <meta charset="UTF-8">
  </head>
  <body>
    <p>A simple paragraph text!</p>
  </body>
</html>
```

Listing 3.3 /examples/chapter003/3_5/index.html

You can specify the relationship between the document and the external document via the HTML attribute `rel`. Possible values for this attribute are listed in Table 3.4. In the example, the document is linked to an external stylesheet file. When using this tag, you should also make sure to specify the MIME type, which I've done here with the `type` attribute and the `text/css` value. However, the most important attribute that you must always use along with the `link` element is the `href` attribute, which specifies the URL of the linked resource. By specifying *style.css*, I assumed the stylesheet file is in the same directory as the HTML document.



A simple paragraph text!

Figure 3.5 Thanks to the Logical Link to the External CSS File, the <p> Element Was Formatted Here in This Example

Specifying a Base URL via the <base> Element

If you want to set the resources from another URL using the `base` element, you still need to put `<base>` before `<link>` in the head area between `<head>` and `</head>`. Example:

```
...
<head>
  <title>Logical linking via link</title>
  <base href="https://css.sap-press.com/">
  <link rel="stylesheet" type="text/css" href="rheinwerk.css">
  <meta charset="UTF-8">
</head>
...
```

Here in `<base>`, *https://css.sap-press.com/* was defined as the base URL for the references, so the reference to the CSS file *https://css.sap-press.com/rheinwerk.css* is added and used in `<link>`. If the `base` element were located behind the `link` element, the *rheinwerk.css* file would again be expected in the same directory as the current document, which may well be intentional. For this reason, you should pay attention to where you define the `base` element because only those references that are located after this element are extended with the base URL.

Regarding the `rel` attribute and its possible values, note that some of these values are only of limited use in practice because how (and whether) a logical link is displayed by the web browser isn't specified. Especially for links with the link types `next`, `prev`, `first`, `last`, `author`, `help`, `search`, `sidebar`, and `license`, it's therefore more advisable (for the time being) to insert a corresponding button or text for this yourself to create a hyperlink (see Chapter 5, Section 5.2). The global `title` attribute enables you to define a caption here that will be displayed if a web browser should support one of these links, that is, the just-mentioned `rel` values such as `next`, `prev`, `first`, and so on.

Attribute	Meaning
`href`	Specifies the URL to the resource to be linked. This attribute must be used.
`hreflang`	Defines the language of the resource to be linked.
`media`	Specifies for which medium/device the target resource in `href` has been optimized. For example, this attribute is popular with stylesheets to define multiple styles for different media types.
`rel`	Sets the relationship or relatedness (the type of link) between the current document and the external resource in `href`. Possible values are as follows: ■ `alternate`: Links to an alternative presentation form of the current page. It's used, for example, to link an RSS or Atom feed to a page. Other similar values are `feed` and `feed alternate`. ■ `author`: Links to another page with information about the author of the current document. You can also use another resource such as a `mailto:` link to an email address of the author. ■ `archives`: links to a previous version of certain documents, such as in a blog archive. ■ `help`: Links to a help document. ■ `icon`: Allows you to assign a favicon to the web page, which will be displayed as a mini graphic in the bookmarks or in the tabs of the browsers. In this context, there are two Apple-specific values for the iPhone or iPad available, `apple-touch-icon` and `apple-touch-startup-icon`, which I'll also explain in Chapter 5, Section 5.2.
`rel`	■ `license`: Links the current page to a page that contains the usage rights for the contents of the current page. ■ `next`, `prev`: Creates a link from the current page to the next (`next`) or previous page (`prev`). ■ `prefetch`: Links an external web page, which probably could be called next by the user, to the current page. This *could* cause the browser to already load this page into a cache even though the user is still viewing the current page. When the user then opens this page, it can load faster from the cache. ■ `pingback`: Specifies the website of a pingback server, which is very useful especially for blogs to handle pingbacks for the current document. ■ `search`: Links the current document to another document where a search across the whole website is possible. ■ `stylesheet`: Probably the most often used and common value for `rel`, as it links an external CSS file to the current document. ■ `tag`: A simple tag as a linked resource that applies to the current document.

Table 3.4 HTML Attributes for the <link> HTML Element

Attribute	Meaning
`size`	Specifies the size(s) for the resource to be linked. Makes sense only if the attribute is `rel="icon"`. Example: `size="16x16"` (one size), `size="16x16 32x32"` (two sizes), or `size="any"` (any size).
`type`	Specifies the MIME type for the document to link to (e.g., `text/css` for a CSS file).

Table 3.4 HTML Attributes for the <link> HTML Element (Cont.)

If you take a closer look at the `rel` values in Table 3.4, you might notice that there are two different types of values here: (1) the attribute values for pure hypertext links, and (2) values for links to external resources. `rel` values to external resources are `icon`, `pingback`, `prefetch`, and `stylesheet`. All other values are pure hypertext links.

Confusion with the "rel" Attribute Values

Besides the `rel` attribute values shown here, you'll probably come across other values on the internet. It's quite hard to keep track of this as well as of what works and what doesn't work on which web browsers. But I'm sure there'll be some movement in this regard in the future. However, you should note that many of those values you find on the web are merely suggestions. Going into this topic in greater depth is beyond the scope of this chapter; however, you can find a good overview and recommendations at *http://microformats.org/wiki/existing-rel-values*. The W3C website also provides a useful overview at *www.w3.org/TR/html5/links.html#sec-link-types*.

3.6 Writing Document-Wide CSS Styles Using <style>

You can use the `style` element to include style information (usually CSS) within the HTML document. Between `<style>` and `</style>`, you define how the web browser should display the HTML elements. Each HTML document can contain multiple `style` elements. Furthermore, since HTML 5, it's valid to use this element within the HTML document body between `<body>` and `</body>`.

Referring to the HTML document */examples/chapter003/3_5/index.html* from Section 3.5, instead of using the `link` element, you could use the `style` element to write the stylesheet information directly in the HTML document as follows:

```
<!doctype html>
<html lang="en">
  <head>
    <style type="text/css">
      p {
        width:220px;
        padding:10px;
        border:5px solid blue;
        margin:0px;
        background-color:#e0ffff;
        text-align:center;
      }
 </style>
    <title>The style element in use</title>
    <meta charset="UTF-8">
  </head>
  <body>
    <p>A simple paragraph text!</p>
  </body>
</html>
```

Listing 3.4 /examples/chapter003/3_6/index.html

As a result, you'll obtain the same image as the one in Figure 3.5. Basically, in this example, only the code from the external CSS file *style.css* was embedded into the HTML document header within the `style` element. The `p` element is again formatted with CSS statements in the example. You don't need to bother about these CSS statements between `<style>` and `</style>` at this stage, as I'll describe CSS in detail later in this book.

Table 3.5 provides an overview of the HTML attributes for the `style` element.

Attribute	Meaning
`media`	Specifies for which medium/device the target resource in `href` has been optimized. This attribute is often used with stylesheets to define multiple styles for different media types.
`type`	Specifies the MIME type for the stylesheet (here, mostly with `text/css` for CSS file).

Table 3.5 Attributes for the <style> Element

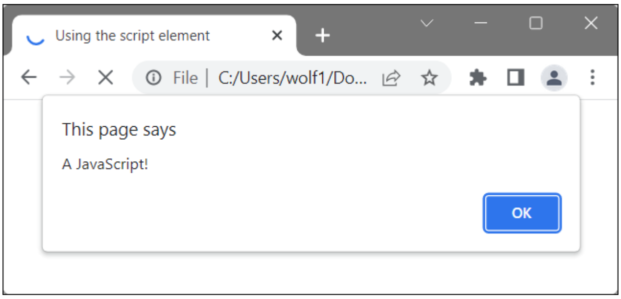
3.7 Including Scripts in Web Pages Using <script>

You can use the `script` element to embed or reference scripts (e.g., JavaScript) in an HTML document. You can either write the script directly between `<script>` and `</script>`, or reference an external script via the `src` attribute, the meaning of which is described in Table 3.6. However, if you want to reference an external script using the `src` attribute, the space between the `<script>` start tag and the `</script>` end tag must be left empty. Unlike the other elements presented here for the head data of an HTML document, you can use the `script` element both in the head section and (multiple times) in the document body.

Here's a simple example of the `script` element, where a simple JavaScript dialog box with the message `A JavaScript!` is displayed on the screen. You can see the example being run in Figure 3.6.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Using the script element</title>
    <script type="text/javascript">
    <!--
      window.onload=alert("A JavaScript!")
    // -->
    </script>
    <meta charset="UTF-8">
  </head>
  <body>
      <p>The first paragraph text!</p>
  </body>
</html>
```

Listing 3.5 /examples/chapter003/3_7/index.html



This page says

A JavaScript!

Figure 3.6 JavaScript (Here, a Simple Dialog Box) Is Executed before the Web Page Gets Displayed

JavaScript Disabled

If the user has JavaScript disabled in the browser or the web browser doesn't support JavaScript at all, you have the option to generate alternative output using `<noscript>`. Anything you write between `<noscript>` and `</noscript>` will be used as an alternative if the browser can't run scripts. Although hardly anyone deactivates JavaScript today and almost every smartphone is perfectly capable of JavaScript, new media are constantly being added that offer browsing the internet, but where only modest and limited browser functionality is available. For example, the latest generations of TVs often have internet features and a web browser built in, but most of the time JavaScript doesn't work.

Let me also describe the use of the `script` element to reference an external JavaScript with the `src` attribute at this point:

```
...
  <head>
    <title>Using the script element</title>
    <script type="text/javascript" src="script.js"></script>
...
  </head>
...
```

Listing 3.6 /examples/chapter003/3_7/index2.html

This example is based on the assumption that a JavaScript named *script.js* is located in the same directory as the HTML document. The script here is usually executed immediately before the web browser continues with the web page. For such external scripts, you can affect the execution time via the `async` and `defer` attributes. Both attributes will be described in greater detail in Table 3.6.

While it's a bit too early for details in JavaScript, it's still worth mentioning here that a script code in the `head` section of an HTML page can increase the loading time because the rest of the page is blocked until the JavaScript has been executed. For this reason, it usually makes more sense to use the script code at the end of the HTML file, most conveniently before the closing `<body>` tag.

Table 3.6 provides an overview of the HTML attributes for the `script` element.

Attribute	Meaning
`async`	If you use `async`, the script gets executed asynchronously with the HTML document. The script is executed while the HTML document is parsed. This attribute can be used only for external scripts.
`charset`	This attribute sets the character encoding for the external script.
`defer`	If you use this attribute, the website gets parsed first and then the script is executed. This attribute can be used only for external scripts.
`src`	This attribute specifies the URL to the external script.
`type`	This attribute enables you to specify the MIME type for the stylesheet (here, mostly with `text/javascript` or `text/ecmascript`).

Table 3.6 Attributes for the <script> Element

3.8 Metadata for the Document Using <meta>

The `meta` element allows you to write additional information or data about the HTML document in the head section between `<head>` and `</head>`. These can be instructions for the web browser, the web server, or a web crawler (also spider, searchbot, bot, or search robot). Even though the use of those `meta` elements is optional, they often get specified. It's quite difficult, especially for beginners, to keep track of the many existing HTML attributes and the possible attribute values you can use with the `meta` element. Many of these additional details aren't standardized at all.

Web Crawler

A *web crawler* is an application that searches the internet and analyzes entire websites. There are different types of web crawlers on the go that collect different types of information. Search engines also use a web crawler to analyze websites. Basically, the principle is quite similar to web browsing, where hyperlinks take you from one web page to other URLs. A web crawler stores these URLs and visits these pages one by one. The websites are evaluated via indexing to make searching for the relevant data possible.

3.8.1 The Most Commonly Used Metadata

A `meta` element is usually composed of at least two attributes. Either the attributes consist of a `name`/`content` combination or an `http-equiv`/`content` combination. In addition, a special version exists for character encoding.

"name/content" Combinations: Freely Definable Metadata

The `meta` element containing the HTML attribute `name` can basically contain any information in the HTML attribute `content`. Theoretically, you could assign any value to the contents of `name` yourself. Nevertheless, some default metadata for the `name` attribute value has been defined in HTML. However, these `name`/`content` combinations aren't intended for personal information, but should only contain information about the HTML document. A simple example might look as follows:

```
...
  <head>
    <title>Freely definable metadata</title>
    <meta name="author" content="John Doe">
    <meta name="keywords" content="metadata, meta, html">
    <meta charset="UTF-8">
  </head>
...
```

Here, you can see two typical `name/content` combinations. The first example defines the author of the web page, while the second pair defines `keywords` for the search engines. You could use any number of other `meta` elements here.

“http-equiv/content” Combinations: HTTP Equivalents

The specifications with `http-equiv` (also called the *pragma directive*) were intended for the web server to communicate. The web server should read this information and then take the read information into account when responding to the client (web browser) and use it in the HTTP response header. However, web servers don’t actually parse HTML documents, so again it’s up to the browser how this information gets processed. Let’s look at a simple example:

```
<!doctype html>
<html lang="en">
  <head>
    <title>HTTP equivalents</title>
    <meta http-equiv="refresh" content="5">
    <meta charset="UTF-8">
  </head>
  <body>
      <p>Page gets refreshed every 5 seconds.</p>
  </body>
</html>
```

Listing 3.7 /examples/chapter003/3_8_1/index.html

The `refresh` value for the `http-equiv` attribute and the value `5` for the `content` attribute allow you to make the web browser refresh the web page every five seconds.

Setting the Character Encoding for the HTML Document

In addition to the `name/content` and `http-equiv/content pairs`, there’s a third option that allows you to specify the character encoding (more easily). Generally, you should use this information when creating a web page that’s written in a language other than English. This is the line with the `meta` element that you use in every example of the book:

```
<meta charset="UTF-8">
```

This will ensure that special characters such as German umlauts and some other special characters are also displayed correctly, thanks to the UTF-8 character set standard. Besides the internet, modern operating systems also use UTF-8, and unless you have a reason to use a different character set, you should always work with UTF-8.

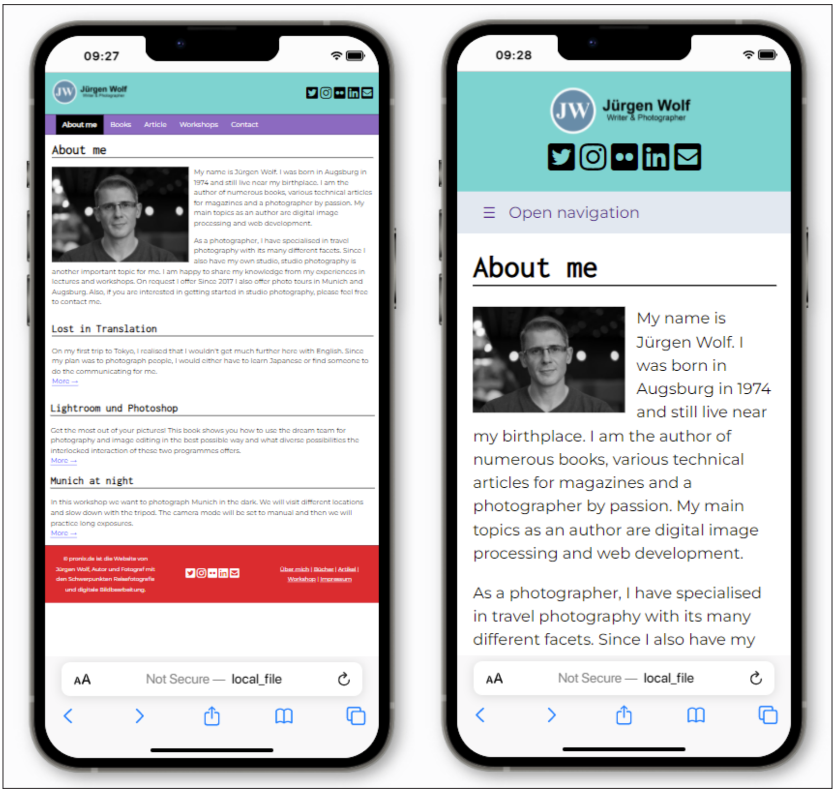
3.8.2 Setting the Viewport

Let's jump ahead to the viewport now, as a correct setting will prevent a responsive website from being displayed in a small view on the mobile device. The viewport is the area of the browser window where the web content gets displayed. Without any special precautions, web pages on a smartphone's mobile browser would be scaled down until they fit completely on the screen. This allows visitors to keep an overview and zoom into the page.

If you want to create modern websites today, then taking into account the different device sizes and a responsive web design is part of the development process. When creating responsive web pages, you must prevent this automatic downsizing. You can do this via a `meta` element like the following:

```
<meta name="viewport" content="width=device-width">
```

It tells the browser to use the actual width of the device rather than an imaginary width. You can see the result of this line in a responsive web page in Figure 3.7, where the automatic resizing function was implemented on the left-hand side and the viewport with the `meta` tag was used on the right.


09:27
Jürgen Wolf
About me
Books
Article
Workshops
Contact
About me
Lost in Translation
Lightroom und Photoshop
Munich at night
AA
Not Secure — local_file
09:28
JW
Jürgen Wolf
Writer & Photographer
☰ Open navigation
AA
Not Secure — local_file

About me

My name is Jürgen Wolf. I was born in Augsburg in 1974 and still live near my birthplace. I am the author of numerous books, various technical articles for magazines and a photographer by passion. My main topics as an author are digital image processing and web development.

As a photographer, I have specialised in travel photography with its many different facets. Since I also have my

Figure 3.7 A Responsive Website: (Left) without a Meta Viewport and (Right) with a Meta Viewport

I'll describe the viewport and responsive web design separately in Chapter 13. Without explaining it in more detail here, the following `meta` element has become accepted for it in the meantime:

```
<meta name="viewport"
      content="width=device-width, initial-scale=1.0, shrink-to-fit=no">
```

By using `initial-scale=1.0`, you can make sure that the browser displays the page with the normal zoom level, and with `shrink-to-fit=no`, you instruct the Safari browser on the iPad not to `shrink` even in split view.

3.8.3 Specifying Useful Metadata for a Web Crawler

This section provides a brief description of some metadata for search engine robots (web crawlers). However, you must be aware that this information is only a recommendation for the web crawlers. Whether the search bots adhere to it is out of your hands. At least these attribute values were partly (co)designed by Google, Yahoo, and Microsoft, so these publishers will probably stick to them. If you want to include information for the web crawler as metadata, you must assign the `robots` value to the `name` attribute. In the `content` attribute, you write (or suggest) what the web crawler has to do when it visits the web page, for example:

```
<meta name="robots" content="index,follow">
```

This allows the search robot to include the web page in the search engine `index` and to `follow` the hyperlinks on the page. However, you can usually omit this information because this is the usual behavior of a web crawler.

If you don't want the page to be indexed or the hyperlinks to be followed, you can use the attribute values `noindex` and/or `nofollow` in `content`:

```
<meta name="robots" content="noindex">
```

Here, you indicate that your website shouldn't be included in the search engine index (`noindex`), so that the page can't be found via a search engine. If you want the page to be included in the search engine index, but don't want the hyperlinks to be followed, you merely need to use the attribute value `nofollow` in `content`.

3.8.4 Useful Metadata for Search Engines

Especially for search engines, two `name` values are important, namely `keywords` and `description`. However, the `keywords` value has lost importance because it was misused in the past to feed search engines with many misleading keywords (*keyword stuffing*) to be listed as close to the top as possible in the search. In the meantime, the search engines are again indexing the content of a website in a more targeted manner and tend to leave the keywords unnoticed (or less noticed). If you still want to specify keywords, you must separate the individual keywords in `content` separated by commas, as the following example shows:

```
<meta name="keywords" content="html, meta, keywords">
```

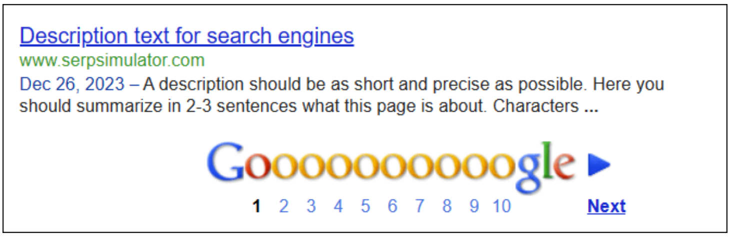
Here, for example, `html`, `meta`, and `keywords` were used as keywords for the website.

What's more interesting, however, is the description text of the website. Although this text will probably not be considered directly in the search results, the description is, in addition to the title, the first thing a user sees listed in the search engine as information from your website. You should keep the description as short and precise as possible and use a maximum of 150 to 250 characters (depending on the search engine). A text that's too long will be shortened.

Here's an example of such a description:

```
...
  <head>
    <title>Description text for search engines</title>
    <meta charset="UTF-8">
    <meta name="description"
          content="A description should be as
                   short and precise as possible. Here
                   you should summarize in 2-3 sentences
                   what this page is about. Characters
                   exceeding the limit will be shortened.">
  </head>
  <body>
   ...
  </body>
...
```

In Google, for example, this description text is usually listed as shown in Figure 3.8.



Description text for search engines
www.serpsimulator.com
Dec 26, 2023 – A description should be as short and precise as possible. Here you should summarize in 2-3 sentences what this page is about. Characters ...

Figure 3.8 Along with the <title> Element, the Description Text Is Often One of the First Features to Appear in a Search Engine

If you don't specify a description with a `meta` element, this text will get generated from the parts of the page content. However, it isn't possible to predict exactly what this description will look like and what kind of text will be used for it. For this reason, you should definitely take the description into your own hands instead of leaving it up to the algorithm of a search engine.

The First Impression Is Important

Although it isn't as important as it was in the early days of the internet, metadata still plays a significant role in search engine coverage. You should therefore always pay attention to the `title` element and the description (`name="description"`) because these elements are often the first things that website visitors get in return from search engines when the page is listed in a search.

3.8.5 Useful Metadata for the Web Browser

If you want to refresh the content of a web page after a certain time or redirect it to another URL, you can use the `http-equiv` attribute with the `refresh` value for this purpose. The `content` attribute enables you to set the time by when the update or redirection should take place.

You can force a refresh of the web page as follows:

```
<meta http-equiv="refresh" content="30">
```

This would refresh the currently loaded web page every 30 seconds.

The redirection to another website can be set up in a similar manner:

```
<meta http-equiv="refresh"
      content="5; URL=http://domain.com/">
```

This causes the browser to switch to the *domain.com* URL after five seconds. You could also use zero seconds here, but this way, you can at least let the user know in the HTML document body why they are being redirected and where.

Stop Using the Automatic Redirection Feature

Automatic redirection can be helpful if the address of the web project has changed. However, some browsers ignore this redirection depending on their settings. In addition, you should also note that the search engines ignore this redirection. In this context, it's often better to define a hyperlink with information in the HTML document body to the new URL with an explanatory note. In addition, when you use the time `0`, it could be difficult for the visitor of the page to use the back button of the browser because this would throw them forward again and again. Alternatively, a redirection can also be created on the server. For example, if you have access to the configuration file *.htaccess* (for Apache web server) or *web.config* (for IIS), you can configure redirecting there. Automatic redirection has been classified as deprecated by the W3C anyway, which is why you should refrain from using it for future web projects. But because redirects are still commonly used, I included the topic here.

As mentioned previously, you can also use the old character encoding specification:

```
<meta http-equiv="content-type"
      content="text/html; charset=utf-8" />
```

This specification corresponds to the more recent specification introduced in HTML:

```
<meta charset="UTF-8">
```

The additional use of the old specification has the advantage that it will also be understood by older browsers that don't know `<meta charset="UTF-8">`.

3.8.6 Using General Metadata

In addition, there's a considerable amount of general metadata, such as the author of the HTML document or the date and time the document was edited. This is helpful, for example, when several people work on one HTML project. You can specify all this information as a `name`/`content` combination. Let's look at some examples:

```
<meta name="author" content="John Doe">
<meta name="date" content=" 2021-01-15T12:00:00+01:00">
```

Here, the author of the web page (`author`) and the date of the last change (`date`) were indicated. If you want to provide personal information for the readers about the current HTML document, you shouldn't do that via metadata, but directly in the HTML document in a readable manner. The metadata is only useful when someone looks at the source code of the document or when it's read by a software. There's also other general metadata such as `generator`, which provides information on the software that was used to create the website. Additionally, you can use `application-name` to make special specifications if the web page belongs to a specific web platform or if a specific web application is running in the web page.

Further Research on the Internet

An overview of the standard metadata can be found on the following website: *www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#standard-metadata-names*. Proposed or future metadata, if any, can be found on this website: *http://wiki.whatwg.org/wiki/MetaExtensions*.

3.8.7 My Recommendation: This Metadata Belongs in the Basic HTML Framework

As you've now been introduced to a number of different types of metadata, you'll probably wonder which type of metadata will be useful for your own website. This is ultimately up to you, but personally I always use at least the character encoding for UTF-8, a page description, and the viewport in the `head` element:

```
...
<head>
    <title>German umlauts</title>
...
    <meta charset="UTF-8" />
    <meta name="description" content="A description should preferably be as
                   short and precise as possible. Here
                   you should summarize in 2-3 sentences
                   what this page is about. Characters
                   exceeding the limit will be shortened."/>
    <meta name="viewport"
          content="width=device-width,initial-scale=1.0, shrink-to-fit=no" />
</head>
...
```

Examples of the Book Remain Shortened

In the examples for the book, I mostly used only the character encoding to keep the source code clearer. The viewport is useful if you create a responsive website, which is what you usually want. The page description is important when you publish the website and want search engines to use the text as a short summary of the contents.

3.8.8 HTML Attributes for the <meta> Element

Table 3.7 provides an overview of the HTML attributes for the `meta` element.

Attribute	Meaning
`content`	Passes the value associated with the attribute of `http-equiv` or `name`.
`charset`	Sets the character encoding for the HTML document.
`http-equiv`	Used for the HTTP response header. For example, you can use it to refresh a web page after a certain time or redirect it to another URL. Possible values: ■ `content-language` ■ `content-type` ■ `default-style` ■ `refresh`

Table 3.7 Attributes for the <meta> Element

Attribute	Meaning
`name`	Defines a name for the metadata. Some default values are as follows: ▪ `application-name` ▪ `author` ▪ `description` ▪ `generator` ▪ `keywords`

Table 3.7 Attributes for the <meta> Element (Cont.)

3.9 Summary

This chapter introduced all the elements you can write in the head of the HTML document between `<head>` and `</head>`. To proceed to the next chapter, it isn't absolutely necessary to memorize all these elements. You can return to some HTML elements at any time if necessary. Following are the two most important elements in this chapter:

- **`title`**
  Just for a valid HTML, you need the `title` element in the head between `<head>` and `</head>` of the HTML document. The `title` element gets displayed in the header bar or tabs of the web browser. It's also used as a suggested name when setting a bookmark and is also listed by search engines as a clickable reference. These arguments should suffice to convince you of the importance of using the `title` element. For the sake of completeness, it's then also recommended to use the page description with `<meta name="description">`.
- **`<meta charset="UTF-8">`**
  You can use the `meta` element to store additional details or data about the HTML document, such as instructions for the web browser, web server, or search engines. While you can use a lot of different specifications here, probably the most important one is the character encoding with `<meta charset="UTF-8">`. Without this information you might run into problems with special characters.

Other elements you've learned about in this chapter include the following:

- The `base` element lets you specify a base URL for all files referenced in the HTML document. By writing such a base URL, you can access a relative or absolute address to the file in the HTML document as if they were on the same computer.
- The `link` element is often used to establish a relationship between the current HTML document and an external document. In practice, it's frequently used, for example, to include a CSS file in the HTML document.
- The `style` element is used to include style information (usually CSS) within the HTML document.
- You can use the `script` element to embed or reference scripts such as JavaScript in the document.

Chapter 4

The Visible Part of an HTML Document

This chapter describes the displayable elements of HTML that you can use between <body> and </body>. For designing or laying out websites, you should use CSS instead. Before you learn how to make a website more beautiful, you need to have the basic knowledge of how to create a single web page using HTML and mark it up with the appropriate elements.

Even if you've already created web pages in HTML 4.01 (or even earlier) and are already familiar with the handling of HTML elements, it's worth working through this chapter because semantic elements have been added with the current HTML and many existing elements have been given a different semantic meaning.

Here's what you'll learn in this chapter:

- Splitting an HTML document into separate and meaningful sections with new HTML elements such as `<section>`, `<article>`, `<aside>`, or `<nav>`
- Using headings in a certain order and implementing a header and/or footer with the new `<header>` and `<footer>` elements
- Splitting and grouping text content with HTML elements
- Semantic tagging of text such as single letters, words, or parts of sentences with HTML elements
- Using and displaying unordered and ordered lists via `<ul>` and `<ol>`

4.1 HTML Elements for Structuring Pages

In this chapter, you'll learn about the various HTML elements that you can use to divide a web page into useful sections. If you've used HTML 4.01 so far, you'll find many new elements here, as the current HTML also introduces a new content model to combat the rampant use of `div` elements with `class` attributes.

HTML Element	Meaning
`<body>`	Displayable content section of the HTML document
`<section>`	Subdivision of the HTML document into different sections

Table 4.1 Quick Overview of the Section Elements Covered Here

HTML Element	Meaning
`<article>`	Subdivision of content into a self-contained topic-specific block
`<aside>`	Marginal information of a content such as a sidebar or for additional information about an article
`<nav>`	Element used to mark up navigation(s) such as a sitemap or the main navigation of the website
`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`	Headings of the first through sixth order
`<header>`	Header of a content
`<footer>`	Footer of a content
`<address>`	Contact information for the author of the content

Table 4.1 Quick Overview of the Section Elements Covered Here (Cont.)

4.1.1 Using <body>: The Displayable Content Section of an HTML Document

Everything you write between the opening `<body>` tag and the closing `</body>` tag is referred to as the *HTML document body*. Between `<body>` and `</body>`, you can write all HTML elements, such as text, hypertext links, images, tables, and lists, to define the structure of the web page. All elements written between `<body>` and `</body>` are rendered by the web browser and displayed accordingly.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Title of the document</title>
    <meta charset="UTF-8">
  </head>
  <body>
    This is the content of the document, which is to be
    rendered and displayed by the web browser.
  </body>
</html>
```

4.1.2 Introducing the Section Elements of HTML

The following sections introduce the section elements of HTML, that is, `<section>`, `<article>`, `<aside>`, and `<nav>`. If you're perhaps just getting into HTML, using section elements is still a bit confusing or disappointing at first because they change almost nothing visually. Primarily, these elements only serve to divide the content into semantic (i.e., meaningful) areas.

Even if these new elements don't seem to make sense to you yet, just remember that they aren't of interest to the normal user of the website, but are mainly used to give meaning to the content, which is particularly useful for the developer, the search engines, and the screen readers.

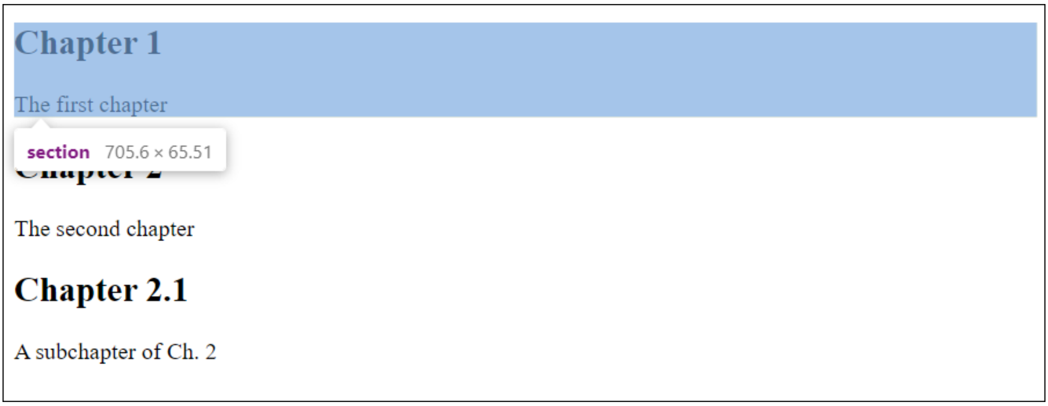
Dividing Content into Topic-Based Sections Using <section>

The `<section>` element allows you to divide the content of a document into topic-based sections. This is helpful, for example, if you want to divide a document into individual chapters or even subchapters—just like this book was divided into individual sections. Even on an ordinary homepage, you can use this element to create individual content and sense sections, such as a section with the description about the owner of the website, another section with news, and one with contact information. Here's a simple example, the result of which you can see in Figure 4.1:

```
...
  <body>
    <section>
       <h1>Chapter 1</h1>
       <p>The first chapter</p>
    </section>
    <section>
        <h1>Chapter 2</h1>
        <p>The second chapter</p>
        <section>
            <h2>Chapter 2.1</h2>
            <p>A subchapter of Ch. 2</p>
        </section>
    </section>
  </body>
...
```

Listing 4.1 /examples/chapter004/4_1_3/index.html

In this example, the `<section>` element has been used to divide the document into meaningful sections—in this case, Chapter 1 and Chapter 2—with each chapter consisting of a heading `<h1>` and paragraph text `<p>`. Furthermore, it's possible to nest `<section>` elements, as shown within the `<section>` element of Chapter 2, Section 2.1.



Chapter 1

The first chapter

Chapter 2

The second chapter

Chapter 2.1

A subchapter of Ch. 2

Figure 4.1 Between <section> and </section>, You Can Divide the Content of a Document into Meaningful and Logical Units

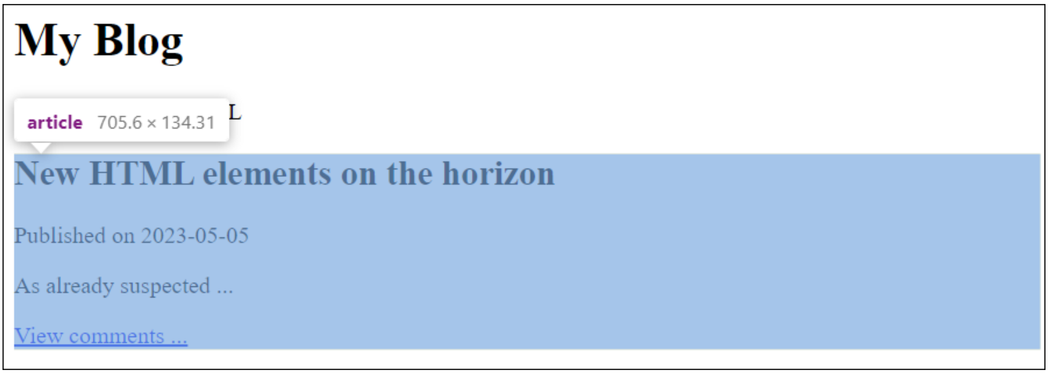
Dividing Content into a Self-Contained Block Using <article>

You should use the `article` element to summarize a piece of content in a self-contained topic-specific block. The `article` element is in itself quite similar to the `<section>` element, which you use to divide the content into meaningful sections. However, it's recommended that you use the `article` element for a standalone composition, which would be ideal for individual news items, blog or forum entries, or comments on a blog post or news, for example.

Here's an example of an HTML code snippet that shows you what such a blog entry with the `article` element could look like. The result is shown in Figure 4.2.

```
...
  <body>
  <h1>My Blog</h1>
  <p>Latest reports on HTML</p>
  <article>
     <header>
       <h2>New HTML elements on the horizon</h2>
     </header>
     <p>Published on <time>2023-05-05</time></p>
     <p>As already suspected ...</p>
     <footer>
       <a href="comments.html">View comments ...</a>
     </footer>
  </article>
  </body>
...
```

Listing 4.2 /examples/chapter004/4_1_4/index.html



My Blog

New HTML elements on the horizon

Published on 2023-05-05

As already suspected ...

View comments ...

Figure 4.2 The Example Shows a Meaningful and Logical <article> Composition of a Blog Entry

Everything between `<article>` and `</article>` is the composition of a self-contained block consisting of a heading, a timestamp, the actual content section, and a footer. It's up to you to decide which HTML elements you want to use to create such a composition with `<article>`, but the example shown here already makes sense semantically.

What to Use: <article> or <section>?

You're probably wondering which of the two elements you should use for a semantic separation of content because the two are somewhat similar in some respects. Nevertheless, the HTML specification also makes a differentiation here and recommends using `<article>` if certain semantics are to be used multiple times, as is the case with a news or blog entry. Thus, `<article>` is a self-contained block—a composition of repeatedly used content following the same pattern—whereas `<section>` is suitable for a separation into content sections, which should contribute to a better overview of the entire document.

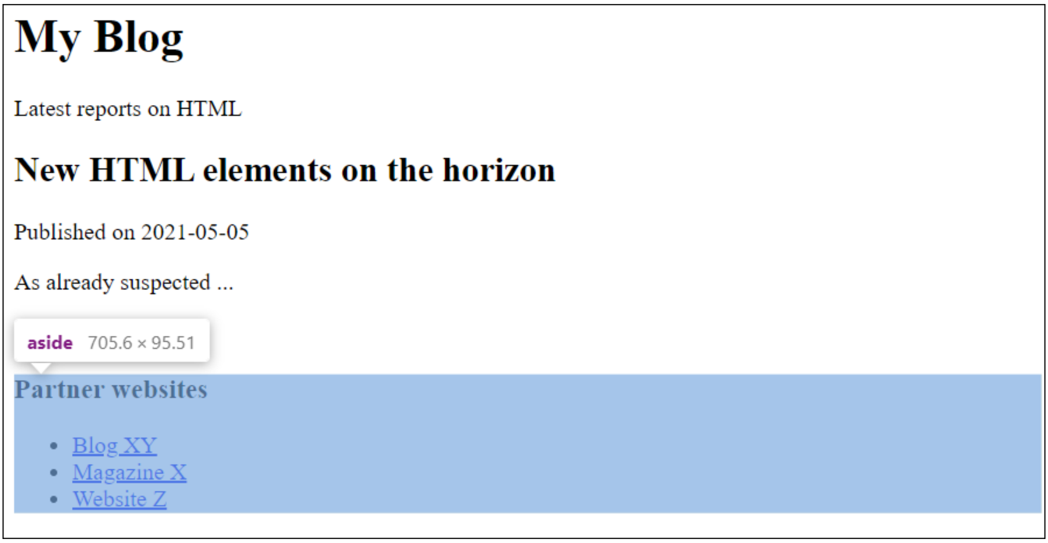
Adding Content with Additional Information Using <aside>

With `<aside>`, you can usually supplement or expand content with additional information. Strictly speaking, you can use the `aside` element for two different semantic things: a sidebar or an additional piece of information (e.g., a citation) to a content item, for example, within an `article` element.

Referring to the */examples/chapter004/4_1_4/index.html* example from the previous section, for example, you would use `<aside>` for a separate logical section in the document:

```
...
  <body>
  <h1>My Blog</h1>
  <p>Latest reports on HTML</p>
  <article>
...
  </article>
  <aside>
  <h3>Partner websites</h3>
      <ul>
      <li><a href="#">Blog XY</a></li>
      <li><a href="#">Magazine X</a></li>
      <li><a href="#">Website Z</a></li>
    </ul>
  </aside>
  </body>
...
```

Listing 4.3 /examples/chapter004/4_1_5/index.html



My Blog

Latest reports on HTML

New HTML elements on the horizon

Published on 2021-05-05

As already suspected ...

Partner websites

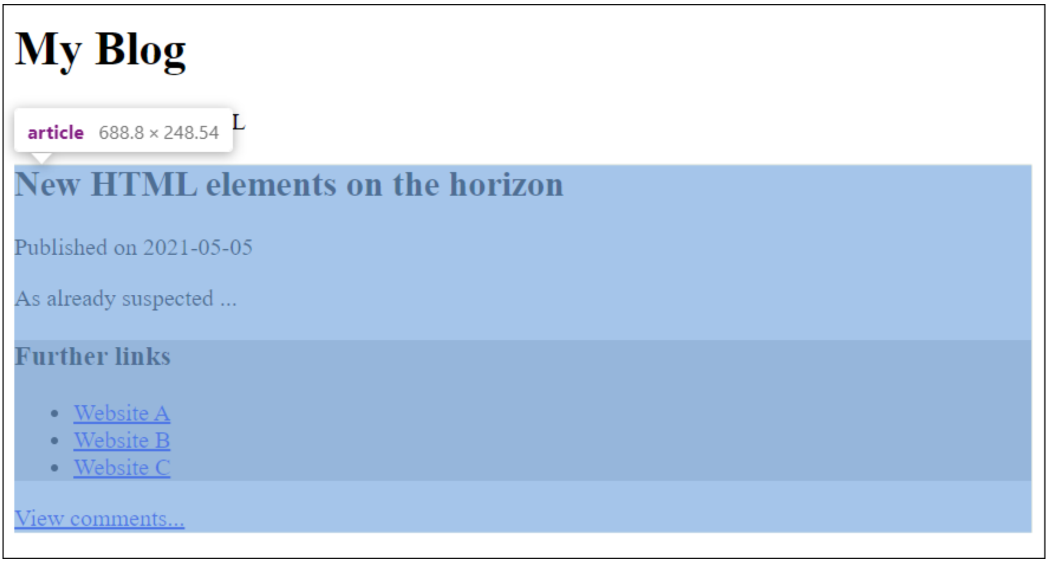
- Blog XY
- Magazine X
- Website Z

Figure 4.3 The <aside> Element Is Used as a Separate Logical Section in the HTML Document

Note

The # character in HTML is a reference to a jump mark in the same document, but it has no meaning yet in this example and was used instead of a real destination address.

In addition to the option just shown, using `<aside>` as a sidebar would also be suitable as additional information in the form of a quote or within an `article` element. In the example that contains the blog entry, it would be suitable within the `article` element for an entry with further links to the blog entry.



My Blog

article 688.8 × 248.54

New HTML elements on the horizon

Published on 2021-05-05

As already suspected ...

Further links

- Website A
- Website B
- Website C

View comments...

Figure 4.4 The <aside> Element (Colored Here) Was Noted as Additional Information inside an <article> Element

Let’s take a look at the following code snippet in this regard:

```
...
  <body>
...
  <article>
    <header>
      <h1>New HTML elements on the horizon</h1>
    </header>
    <p>Published on <time>2023-05-05</time></p>
    <p>As already suspected ...</p>
    <aside>
    <h3>Further links</h3>
      <ul>
        <li><a href="#">Website A</a></li>
        <li><a href="#">Website B</a></li>
        <li><a href="#">Website C</a></li>
      </ul>
    </aside>
    <footer>
      <a href="comments.html">View comments...</a>
    </footer>
  </article>
  <aside>
...
```

```
  </aside>
  </body>
...
```

Listing 4.4 /examples/chapter004/4_1_5/index2.html

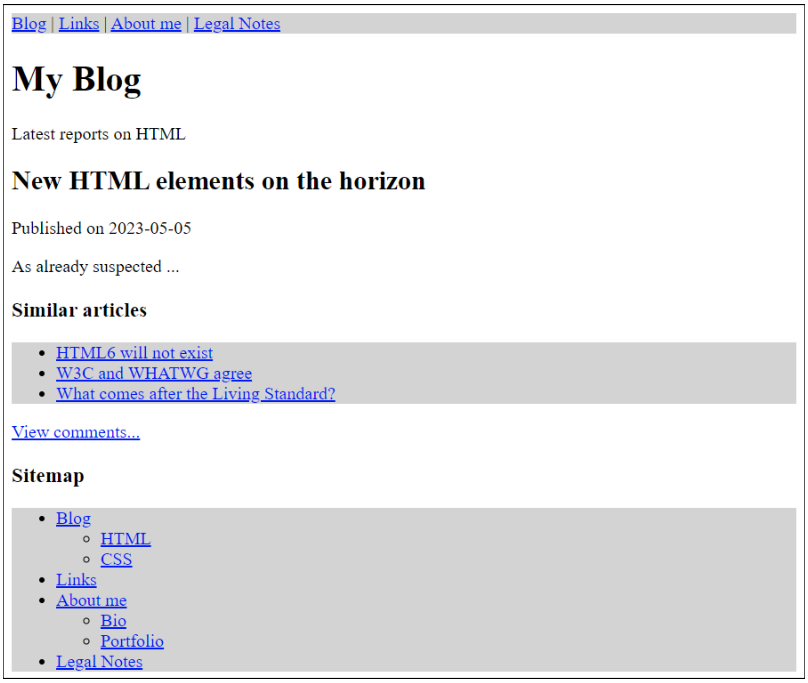
Declaring Content as a Page Navigation Bar Using <nav>

As you might guess from its name, the `nav` element enables you to divide navigation elements into blocks. We're not talking about web link collections here, but about a list of links for a sitemap or the main navigation of your own website. Like the `aside` element, you can use the `nav` element for its own section or within another HTML element to combine a group of links into a block.

To use the blog entry again as an example, the `nav` element would be suitable for summarizing the main navigation or the list of related links from similar articles within the same web page. In any case, you should use the `nav` element for entire blocks of links. The following code snippet demonstrates the `nav` element in a small theoretical blog:

```
...
  <body>
  <nav>
    <a href="#">Blog</a> |
    <a href="#">Links</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </nav>
  <h1>My Blog</h1>
  <p>Latest reports on HTML</p>
  <article>
...
    <aside>
    <h3>Similar articles</h3>
      <nav>
        <ul>
          <li><a href="#">HTML6 will not exist</a></li>.
          <li><a href="#">W3C and WHATWG agree</a></li>
          <li><a href="#">What comes after the Living Standard?</a></li>
        </ul>
      </nav>
    </aside>
...
  </article>
  <aside>
  <h3>Sitemap</h3>
    <nav>
    <ul>
      <li><a href="#">Blog</a>
        <ul>
          <li><a href="#">HTML</a></li>
          <li><a href="#">CSS</a></li>
        </ul>
      </li>
      <li><a href="#">Links</a></li>
      <li><a href="#">About me</a>
        <ul>
          <li><a href="#">Bio</a></li>
          <li><a href="#">Portfolio</a></li>
        </ul>
      </li>
      <li><a href="#">Legal Notes</a></li>
    </ul>
    </nav>
  </aside>
  </body>
...
```

Listing 4.5 /examples/chapter004/4_1_6/index.html



Blog | Links | About me | Legal Notes

My Blog

Latest reports on HTML

New HTML elements on the horizon

Published on 2023-05-05

As already suspected ...

Similar articles

- HTML6 will not exist
- W3C and WHATWG agree
- What comes after the Living Standard?

View comments...

Sitemap

- Blog
  - HTML
  - CSS
- Links
- About me
  - Bio
  - Portfolio
- Legal Notes

Figure 4.5 The <nav> Element (Colored Here) Can Be Used to Divide a Separate (Navigation) Section or to Group Blocks of Links within Other HTML Elements

In the first example, `<nav>` was used to define a main navigation as a separate section of the HTML document. In the second example, the `nav` element was used to link a block of links to similar articles on the same web page. In the last example, a sitemap of the web page was summarized via the `nav` element.

In addition, the last two examples were grouped within `<aside>` and `</aside>`. In them, bulleted lists (`ul` and `li` elements) were used within the `nav` element.

Using <nav> Only for Main Navigation?

The specification suggests using the `nav` element specifically for the main navigation. That doesn't include external additional links or affiliate links to external websites. Likewise, it's not recommended to put legal stuff such as copyright, contact information, and legal notes in the `nav` section; instead, use the `footer` section for that purpose (see Section 4.1.4).

In the */examples/chapter004/4_1_6/index.html* example, this means you only have two main navigation points with `Blog` and `Links` within the `nav` element and would write `About me` and `Legal Notes` outside of it (e.g., in the footer). I don't see any point in separating this because there's no difference between the first two links (**Blog** and **Links**) and the other two links (**About me** and **Legal Notes**) as both are linked to different websites, and both belong to the internal website in this case. So, it's up to you to what extent you want to follow these recommendations. In any case, you should refrain from using the `nav` element for external links to third-party websites and, if possible, use it only selectively and sensibly on your own website.

4.1.3 Using Headings with the HTML Elements from <h1> to <h6>

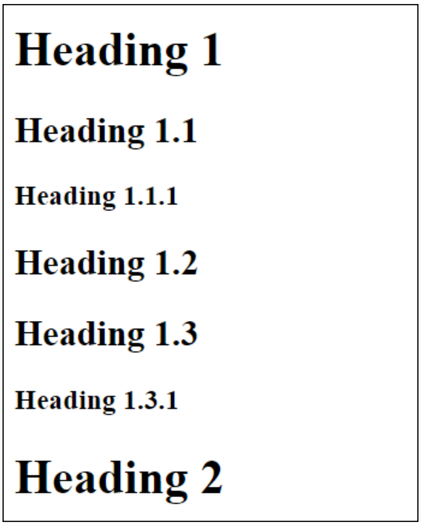
The HTML element for headings of a certain order is `<h1>` to `<h6>`. The number (1 to 6) represents the heading level. Thus, everything you write between `<h1>` and `</h1>` is used as a top-level heading, everything between `<h2>` and `</h2>` belongs to a second-level heading, and so on down to the lowest level with `<h6>` and `</h6>` as a sixth-level heading.

The HTML elements `<h1>` through `<h6>` should not be misused to emphasize a text, but rather to define the content structure of a document. Consider the following HTML structure:

```
...
  <h1>Heading 1</h1>
  <h2>Heading 1.1</h2>
  <h3>Heading 1.1.1</h3>
  <h2>Heading 1.2</h2>
  <h2>Heading 1.3</h2>
  <h3>Heading 1.3.1</h3>
  <h1>Heading 2</h1>
...
```

Based on this sequence of headings, the following content structure (or *document outline*) will be mapped:

```
1. Heading 1
   1.1. Heading 1.1
      1.1.1. Heading 1.1.1
   1.2. Heading 1.2
   1.3. Heading 1.3
      1.3.1. Heading 1.3.1
2. Heading 2
```



Heading 1

Heading 1.1

Heading 1.1.1

Heading 1.2

Heading 1.3

Heading 1.3.1

Heading 2

Figure 4.6 This Is What the Web Browser Will Make of It

What Happens to the Headings in the Section Elements?

You're probably wondering what happens to the content structure of headings when you use the section elements from Section 4.1.2. That question is well justified. If you use the `<section>`, `<article>`, `<aside>`, or `<nav>` section elements, the content structure of the headings will also be affected. Within each new section element, the heading level count starts from the beginning, but always at a lower hierarchy level. The following HTML code illustrates this:

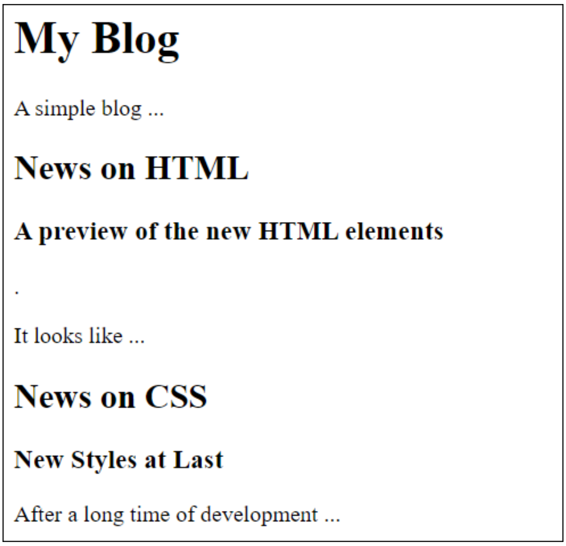
```
...
  <body>
  <h1>My Blog</h1>
  <p>A simple blog ...</p>
  <section>
```

```
    <h1>News on HTML</h1>
    <article>
      <h1>A preview of the new HTML elements</h1>
      <p>It looks like ...</p>
    </article>
  </section>
  <section>
    <h1>News on CSS</h1>
    <article>
      <h1>New Styles at Last</h1>
      <p>After a long time of development ...</p>
    </article>
  </section>
  </body>
...
```

Listing 4.6 /examples/chapter004/4_1_7/index.html

Here, five `<h1>` headings of the first order were used. If you look at the HTML code, you can see several sections. Next to the top section with `<body>`, you can find two additional `<section>` elements, each of which contains an `<article>` element in which headings of the first order have also been defined.

This is a blog that's been divided into two content sections with `<section>` containing the topics HTML and CSS. Within these sections, you can find the news articles included within `<article>`.



My Blog

A simple blog ...

News on HTML

A preview of the new HTML elements

.

It looks like ...

News on CSS

New Styles at Last

After a long time of development ...

Figure 4.7 All Headings with <h1> Are Adjusted and Output Corresponds to the Section due to the Section Elements of HTML That Are Based on the Outline Algorithm

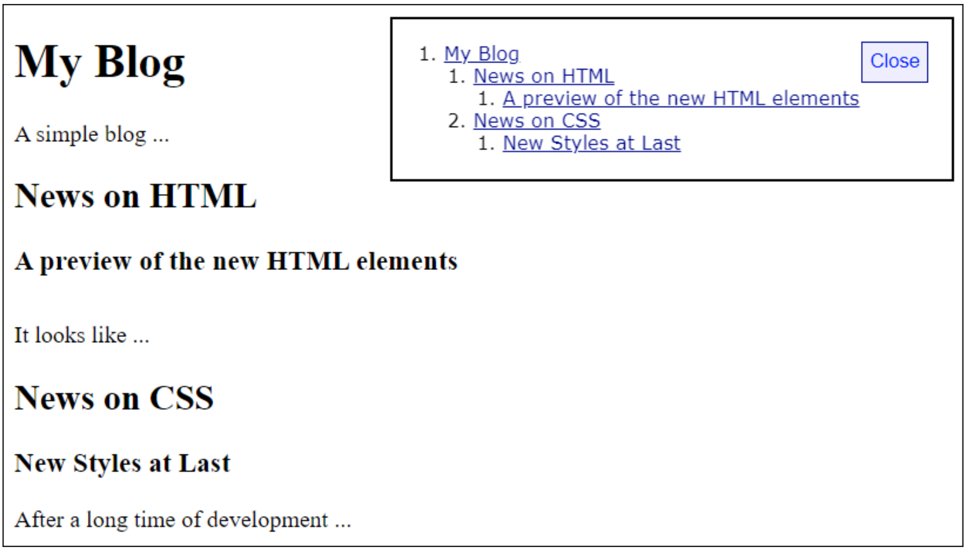
Due to the use of the new HTML elements `<article>` and `<section>`, the following content structure (or document outline) results:

```
1. My Blog
   1. News on HTML
      1. A preview of the new HTML elements
   2. News on CSS
      1. New Styles at Last
```

Document Outline for Advanced Users

The term *outline* or *document outline* refers to the structure of the document, which can be generated and represented by the headings, among other things—as in the case of the table of contents of this book, for example. The document outline can be quite useful. For example, the web browser might offer you a table of contents, letting you jump from one heading to another. Search engines can also use such a table of contents to create better page previews or even improve search results. Screen reader users probably have the biggest advantage here because they can be guided through deeply nested hierarchies and sections.

In Figure 4.8, you can see the JavaScript HTML5 outliner (h5o) to test the document outline during execution. Here, the document outline is displayed in the upper-right corner, and you can jump to the individual headings via hypertext links.



My Blog

A simple blog ...

News on HTML

A preview of the new HTML elements

It looks like ...

News on CSS

New Styles at Last

After a long time of development ...

1. My Blog
   1. News on HTML
      1. A preview of the new HTML elements
   2. News on CSS
      1. New Styles at Last

Figure 4.8 JavaScript h5o from Google during Execution

Section elements such as `<section>`, `<article>`, `<aside>`, and `<nav>` allow you to refine the document outline even more, as you've seen in the example, */examples/chapter004/4_1_7/index.html*.

Even if not all web browsers support document outlines directly, it won't do any harm to pay attention to a proper outline of the HTML document because basically that's no extra work. Screen reader users will thank you for it, and search engine robots may reward you for it because a good document outline can improve a page index, which in turn could mean a higher ranking in search results. Besides, a neatly structured web page is easier to read than an unstructured one.

Keeping Track of the Document Outline

When the website becomes more extensive and the document contains many headings and perhaps different sections, it often isn't easy to keep track of whether the document outline still makes sense and is neatly structured with regard to its contents. Outlining tools that output the headline structure of the web page in the existing structure can assist you here. For example, Google offers the JavaScript h5o, mentioned in the previous section, at *https://h5o.github.io*. Alternatively, you can find an online service at *http://gsnedders.html5.org/outliner/*. Meanwhile, the validation checker at *https://validator.w3.org/nu/#textarea* also provides an outline option for HTML documents.

4.1.4 Creating a Header Using <header> and a Footer Using <footer>

The `<header>` and `<footer>` are two additional semantic HTML elements that you can use for implementing a header and footer in an HTML document. Like section elements, these elements initially have no visual effect on the HTML document apart from a line break. Again, these are initially just elements that you can use to give a piece of content a better and cleaner structure. The styling here is usually done via CSS. However, unlike section elements such as `<section>`, `<article>`, `<aside>`, or `<nav>`, these two elements don't affect the hierarchical structuring (or document outline) of the document.

You should use the `header` element for introductory elements such as a page heading, the name of the web page, or a navigation bar of the HTML document. There may well be other HTML elements between `<header>` and `</header>`. However, you mustn't nest any other `header` elements in it. Although it seems obvious, `<header>` doesn't necessarily have to be in the header, and you can use it more than once in the document.

Invalid Positions of <header>

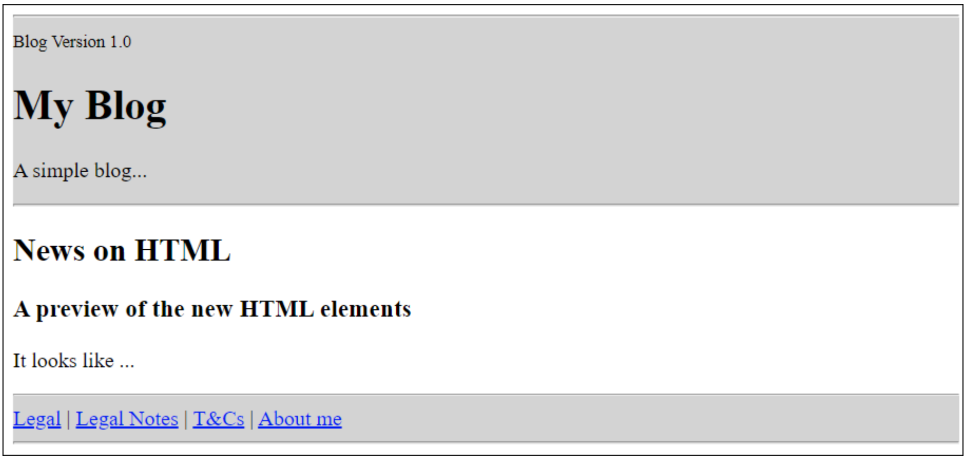
A `<header>` tag mustn't be used inside a `footer`, `address`, or another `header` element.

The counterpart to the `header` element for the header section is the `footer` element for the footer or also the footer section, which also doesn't necessarily have to be the last element in the document. Useful content for the footer of a website is often legal information, legal notes, or terms and conditions, but you can also use a sitemap or a special navigation bar here. You can't use any other `<footer>` tag within a `footer`

Here's an example that demonstrates the `header` and `footer` elements in a meaningful structure:

```
...
  <body>
  <header>
    <hr /><small>Blog Version 1.0</small>
    <h1>My Blog</h1>
    <p>A simple blog...</p><hr />
  </header>
  <h2>News on HTML</h2>
  <article>
    <h3>A preview of the new HTML elements</h3>
    <p>It looks like ...</p>
  </article>
  <footer>
    <hr /><a href="#">Legal</a> |
    <a href="#">Legal Notes</a> |
    <a href="#">T&Cs</a> |
    <a href="#">About me</a><hr />
  </footer>
  </body>
...
```

Listing 4.7 /examples/chapter004/4_1_8/index.html



Blog Version 1.0

My Blog

A simple blog...

News on HTML

A preview of the new HTML elements

It looks like ...

Legal | Legal Notes | T&Cs | About me

Figure 4.9 The Header and Footer with the <header> and <footer> Elements (Shown in Gray for Clarity)

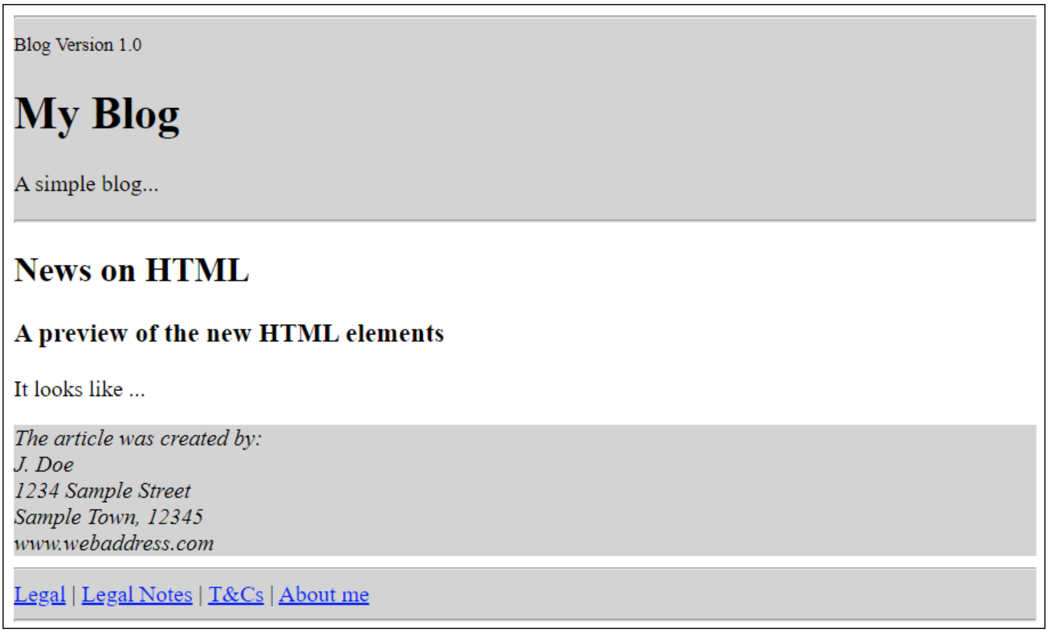
Between `<header>` and `</header>`, you can find a summary of the entire information for the header section of a web page. In the example, this is the version of the blog, the headline, and a short description of the website. This is followed by the articles of the blog. Finally, the footer between `<footer>` and `</footer>` contains forwarding hyperlinks with legal information and so on.

4.1.5 Marking Contact Information Using <address>

You should use the `address` element only for contact information about the author of the HTML document or article. If the `address` element is used within the `body` element, it should only contain the contact information for the owner or author of the entire document or article. If the `address` element is positioned inside an `article` element, the contact information for the author of the document should be written there. Usually, the web browser displays this text in italics with a new line before and after the `address` element.

The best location for this contact information for the author, an organization, or the person responsible for the document or article is usually likely to be at the end of the article or at the end of the document (e.g., between `<footer>` and `</footer>`).

Here's an example in which the `address` element was used for contact information about the author of an article at the end inside the footer element. You can see the example at execution in Figure 4.10.



Blog Version 1.0

My Blog

A simple blog...

News on HTML

A preview of the new HTML elements

It looks like ...

The article was created by:
J. Doe
1234 Sample Street
Sample Town, 12345
www.webaddress.com

Legal | Legal Notes | T&Cs | About me

Figure 4.10 Contact Information for the Author of the Article Has Been Placed at the End of the Article between <footer> and </footer> Using the <address> Element

```
...
  <article>
    <h3>A preview of the new HTML elements</h3>
    <p>It looks like ...</p>
    <footer>
      <address>The article was created by:<br>
        J. Doe<br>
        1234 Sample Street<br>
        Sample Town, 12345<br>
        www.webaddress.com
      </address>
    </footer>
  </article>
...
```

Listing 4.8 /examples/chapter004/4_1_9/index.html

4.2 HTML Elements for Structuring Text

This section describes the HTML elements for grouping or structuring plain text content, such as paragraph text or a line break. This has nothing to do with dividing an HTML document into individual sections or areas. You've previously learned how to do that in Section 4.1.

HTML Element	Meaning
`<p>`	Text paragraph
`<br>`	Forcing a line break
`<wbr>`	Optional line break within a word
`<hr>`	Topic-based separation at the paragraph level
`<blockquote>`	Citation as a text paragraph
`<div>`	Defining a general section
`<main>`	Used for the main content area of a web page
`<figure>`	Grouping or summarizing content for separate description
`<figcaption>`	Labeling content grouped via the `figure` element
`<ul>`	Unordered bulleted list
`<ol>`	Ordered list (mostly numbered)

Table 4.2 Brief Overview of the Elements Covered Here for Grouping and Dividing Content

HTML Element	Meaning
`<li>`	List element in a `ul` or `ol` list
`<dl>`	Creating a description list using `dt` and `dd`
`<dt>`	Expression to be described before the `dd` element
`<dd>`	Description that follows the `dt` element

Table 4.2 Brief Overview of the Elements Covered Here for Grouping and Dividing Content (Cont.)

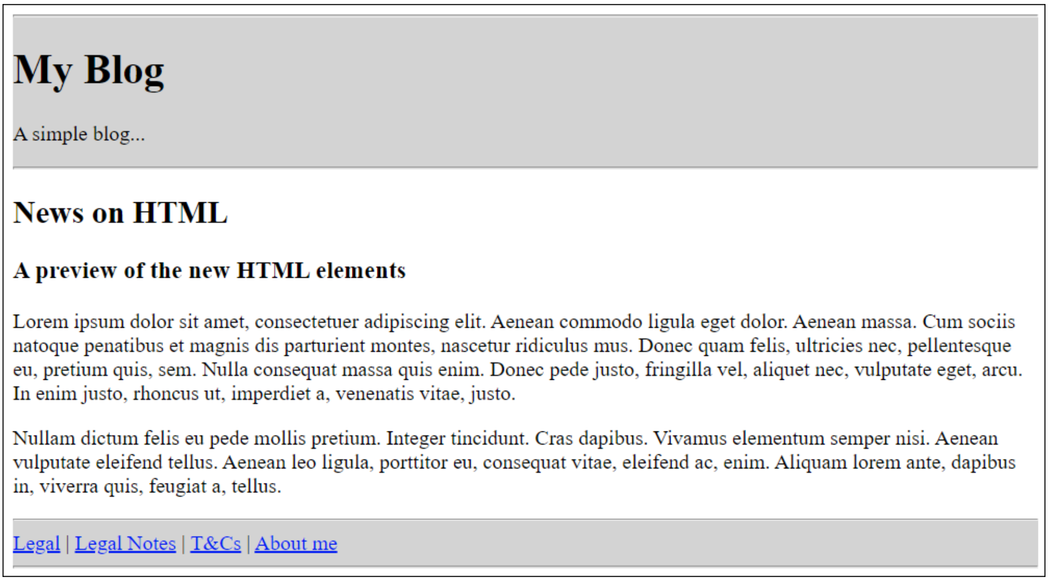
4.2.1 Adding Text Paragraphs Using <p>

The `p` element (`p` = *paragraph*) is the classic element for text paragraphs in a longer continuous text. Anything you write here between the opening `<p>` and the closing `</p>` is treated as a text paragraph. Within such a text paragraph you can use images, videos, audio clips, or other text markup in addition to multiline body text. However, you can't use other group elements, headings(`<h1>` to `<h6>`), or section elements within `<p>` and `</p>`.

The following example demonstrates two slightly longer paragraph texts with the `p` element in use:

```
...
  <body>
  ...
  <h2>News on HTML</h2>
  <article>
    <h3>A preview of the new HTML elements</h3>.
    <p>Lorem ipsum dolor sit amet, consectetuer adipiscing
       elit. Aenean commodo ligula eget dolor. Aenean massa.
       Cum sociis natoque penatibus et magnis dis parturient
       montes, nascetur ridiculus mus. Donec quam felis,
       ultricies nec, pellentesque eu, pretium quis, sem.
       Nulla consequat massa quis enim. Donec pede justo,
       fringilla vel, aliquet nec, vulputate eget, arcu. In
       enim justo, rhoncus ut, imperdiet a, venenatis vitae,
       justo.
    </p>
    <p>Nullam dictum felis eu pede mollis pretium. Integer
       tincidunt. Cras dapibus. Vivamus elementum semper
       nisi. Aenean vulputate eleifend tellus. Aenean leo
       ligula, porttitor eu, consequat vitae, eleifend ac,
       enim. Aliquam lorem ante, dapibus in, viverra quis,
       feugiat a, tellus.
    </p>
  </article>
  ...
  </body>
...
```

Listing 4.9 /examples/chapter004/4_2_1/index.html



My Blog

A simple blog...

News on HTML

A preview of the new HTML elements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus.

Legal | Legal Notes | T&Cs | About me

Figure 4.11 Two Paragraphs with Body Text between `<p>` and `</p>` Displayed in the Web Browser

Aligning and Formatting Paragraph Text Using CSS

Paragraph text with the `p` element can be formatted using CSS or CSS features.

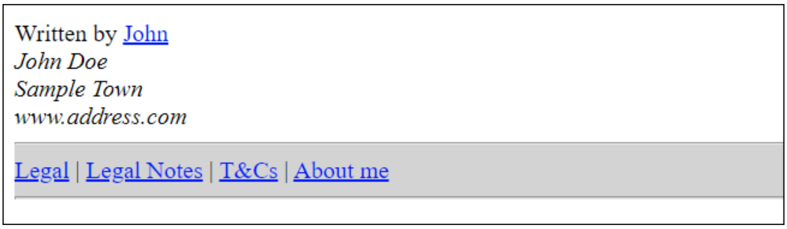
4.2.2 Forcing Line Breaks Using <br>

If you try to insert a line break or a space in the body text of the example just shown, */examples/chapter004/4_2_1/index.html*, you'll notice that it doesn't work. The point at which the line break is supposed to be inserted is decided by the web browser based on a space that separates words. Nevertheless, you can also force a line break at a certain point in the text using `<br>` (`br` = *break*). `<br>` is a standalone tag. Even though you can use multiple line breaks simultaneously via `<br>`, you shouldn't overuse it for separating paragraphs.

The following example is commonly used to represent an address neatly by means of forced line breaks (see Figure 4.12):

```
...
    Written by <a href="mailto:#">John</a><br>
    <address>
       John Doe<br>
       Sample Town<br>
       www.address.com
    </address>
...
```

Listing 4.10 /examples/chapter004/4_2_2/index.html



Written by John
John Doe
Sample Town
www.address.com

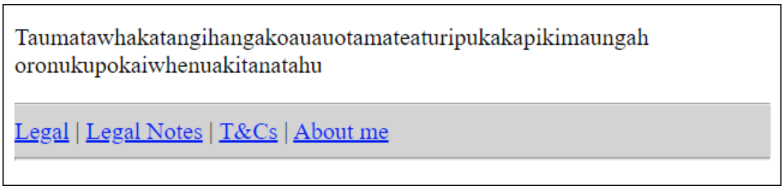
Figure 4.12 You Can Force Line Breaks via the <br> Element

4.2.3 Adding Optional Line Breaks Using <wbr>

If, on the other hand, you need an optional line break that only occurs at a specific position when it's necessary for an optimal display in the web browser and to save the user from scrolling sideways, you can use the standalone `<wbr>` (or `<wbr />` in XHTML) tag for this (`wbr` = *word break*). `<wbr>` can be quite useful if you want to prevent the web browser from breaking a line in the wrong place. A simple example follows:

```
<p>Taumatawhakatangi<wbr>
hangakoauauotamatea<wbr>
turipukakapikimaungah<wbr>
oronukupokaiwhen<wbr>
uakitanatahu</p>
```

Depending on how wide the display section is in the web browser, the long word can be wrapped only at the places where `<wbr>` was inserted.



Taumatawhakatangihangakoauauotamateaturipukakapikimaungah
oronukupokaiwhenuakitanatahu

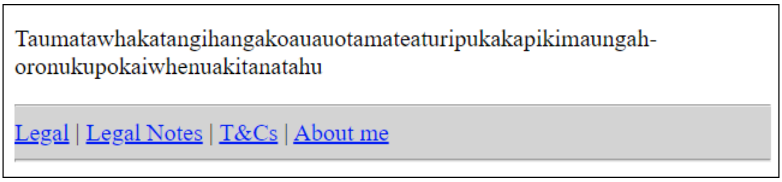
Figure 4.13 An Extremely Long Word Wrapped at a Position Suggested by <wbr>

Line Break via <wbr>

The optional line break via `<wbr>` was added to the standard HTML version at a later time, but it had been around since HTML 2.0. The Netscape web browser had introduced this element a long time ago, and other browser manufacturers had implemented it as well.

Nevertheless, this word separation is unattractive because the line gets broken without consideration of a grammatically correct separation. As an alternative, the *named entity* `&shy;` is suitable for a conditional hyphen. Similar to `<wbr>`, this allows the web browser to separate the word at this point if necessary. Unlike `<wbr>`, the web browser adds a hyphen at the end of the wrapped word. The alternative example looks as follows:

```
<p>Taumatawhakatangi&shy;
hangakoauauotamatea&shy;
turipukakapikimaungah&shy;
oronukupokaiwhen&shy;
uakitanatahu</p>
```



Taumatawhakatangihangakoauauotamateaturipukakapikimaungah-
oronukupokaiwhenuakitanatahu

Figure 4.14 A Long Word Can Also Be Wrapped at the Position Suggested by “&shy;” but It Also Adds a Separator, Unlike <wbr>

4.2.4 Forcing Spaces and Preventing Wrapping Using "&nbsp;"

If you want to insert multiple spaces between two words, you can force this with the named character `&nbsp;` (`&nbsp` = *nonbreaking space*). Let’s take a look at a simple example:

```
<p>word1&nbsp;&nbsp;&nbsp;&nbsp;word2</p>
<p>word3    word4</p>
```

Between `word1` and `word2`, the named HTML entity character `&nbsp;` has been written four times, resulting in four spaces between these two words, which is also displayed by the web browser. In the example with `word3` and `word4` the same was tried by pressing the spacebar four times. Nevertheless, in this case, the web browser displays only one space between the words.

In addition to forcing a space, you can also use the named entity character `&nbsp;` to prevent a break between two words, which the web browser automatically performs when there is insufficient space at the end of a line. For example:

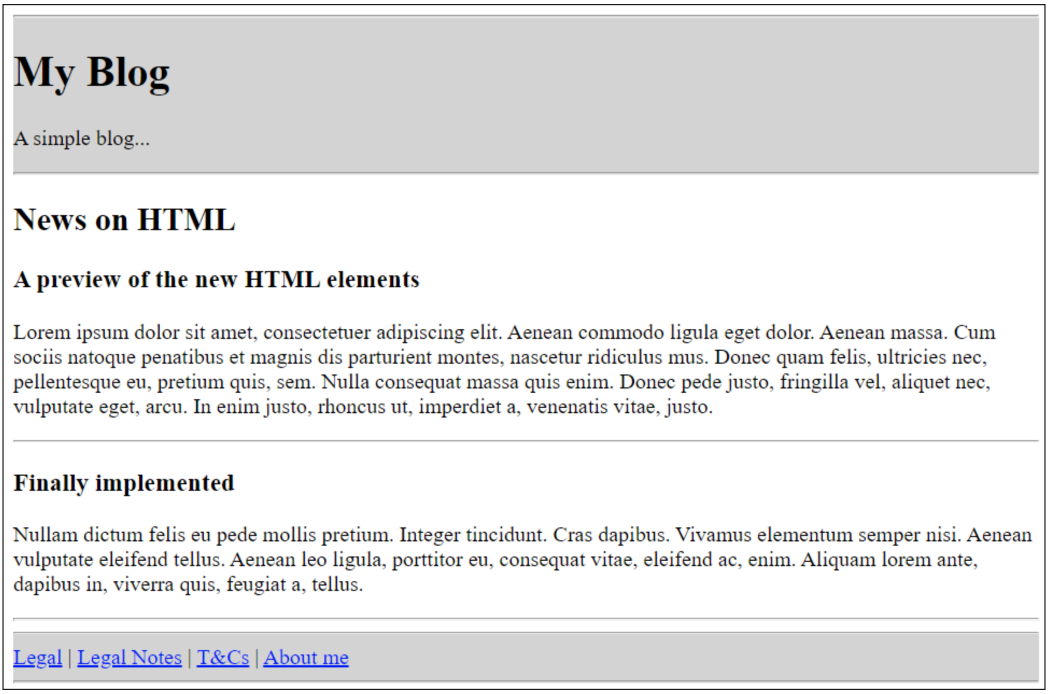
```
... word1&nbsp;word2 ...
```

By placing this `&nbsp;` between `word1` and `word2`, you can prevent the two words from being split between two different lines by the web browser if there's a lack of space. `word1` and `word2` thus stick together in the same line forever.

4.2.5 Adding a Topic-Based Separation Using <hr>

You can use `<hr>` to create a topic-based separation in an HTML document, for example, to separate content more clearly. However, even though `<hr>` is visualized as a separator in HTML by web browsers, the element is also to be treated as a semantic element and not a presentation element. For example, it isn't valid HTML to use the `hr` element between `<p>` and `</p>` or within a heading (`<h1>` to `<h6>`), even though web browsers are quite fault-tolerant about this.

The example shown in Figure 4.15 with a horizontal line can be found under */examples/chapter004/4_2_5/index.html*. You'll see that a separator line also creates a paragraph.



My Blog

A simple blog...

News on HTML

A preview of the new HTML elements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

Finally implemented

Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus.

Legal | Legal Notes | T&Cs | About me

Figure 4.15 With <hr>, a Visual Topic-Based Separation Has Been Added as a Separator Line behind the Paragraph Text

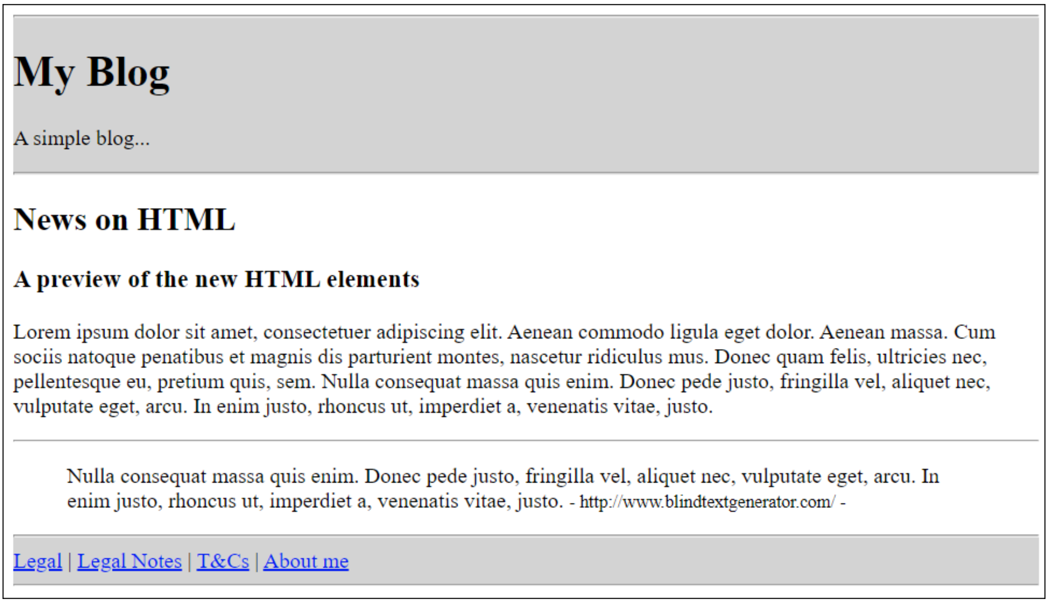
4.2.6 Adding Paragraphs or Citations Using <blockquote>

Between `<blockquote>` and `</blockquote>`, you can quote a text from another source. Most web browsers indent the text in a new paragraph. Within such block quotes, you can use other HTML elements besides text.

The `blockquote` element contains `cite`, an HTML attribute that allows you to specify the source of the citation. With regard to books, this can also be a link to the corresponding book page or to a store where this book can be purchased. Unfortunately, no web browser provides the option to somehow make this source visible or to call the corresponding URL yet. So, to be on the safe side, you should add the source, as I did in the following example; in Figure 4.16, you can see the display in the web browser.

```
...
<blockquote cite="http://www.blindtextgenerator.com/">
   Nulla consequat massa quis enim. Donec pede justo,
   fringilla vel, aliquet nec, vulputate eget, arcu. In enim
   justo, rhoncus ut, imperdiet a, venenatis vitae, justo.
   <small> - http://www.blindtextgenerator.com/ - </small>
</blockquote>
...
```

Listing 4.11 /examples/chapter004/4_2_6/index.html



My Blog

A simple blog...

News on HTML

A preview of the new HTML elements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

> Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. - http://www.blindtextgenerator.com/ -

Legal | Legal Notes | T&Cs | About me

Figure 4.16 Text Quoted between <blockquote> and </blockquote> from the www.blindtextgenerator.com Website

4.2.7 Defining a General Section Using <div>

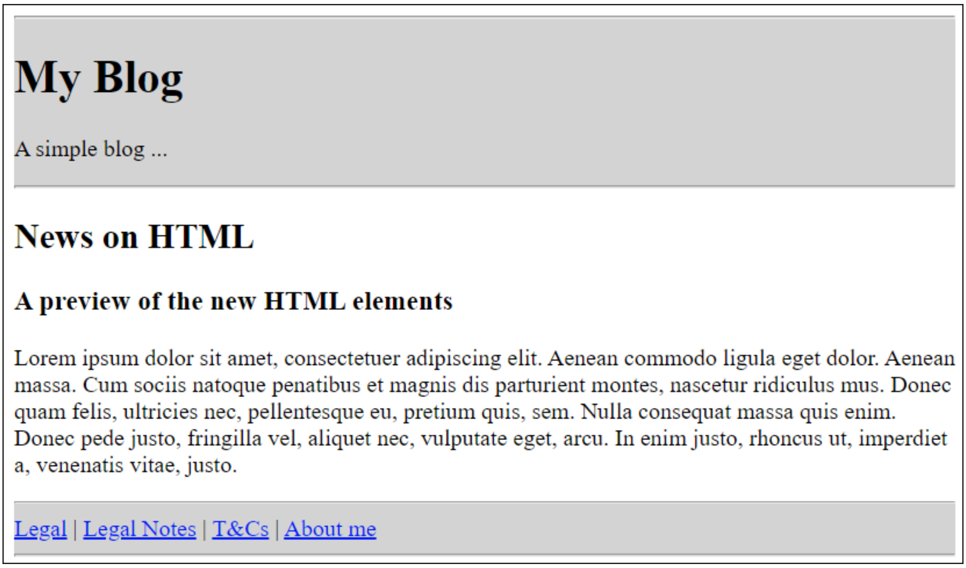
Between `<div>` and `</div>` (`div` = *division*), you can define a general section, which at first usually does nothing but create a new line. This `div` element doesn't have any meaning until CSS comes into play, which is the main use of `<div>`: defining layout sections. In the following example, the HTML attribute `class` was used, which you can use to assign the `div` elements to a class that you can later select with CSS (using a selector) and visually customize or style. Here's a familiar example that demonstrates such an application in use:

```
...
  <body>
  <div class="header">
    <hr />
    <h1>My Blog</h1>
    <p>A simple blog ...</p>
    <hr />
  </div>
  <h2>News on HTML</h2>
  <div class="article">
    <h3>A preview of the new HTML elements</h3>
    <p>Lorem ipsum dolor ...</p>
  </div>
  <div class="footer">
    <hr />
    <a href="#">Legal</a> |
    <a href="#">Legal Notes</a> |
    <a href="#">T&Cs</a> |
    <a href="#">About me</a>
    <hr />
  </div>
  </body>
...
```

Listing 4.12 /examples/chapter004/4_2_7/index.html

For such examples, you should prefer semantic elements such as `<header>`, `<footer>`, `<article>`, `<nav>`, and so on instead of the `div` element.

Therefore, you should use the `div` element only if no other suitable HTML element is available. You can find more information about this in greater detail in Section 4.3. In regard to the */examples/chapter004/4_2_7/index.html* example, you should, as previously described in the book, use the HTML elements `<header>`, `<article>`, and `<footer>` that have been newly introduced in HTML instead of the `<div class="header">`, `<div class="article">`, and `<div class="footer ">` sections used in the previous example. The corresponding example thus looks as follows (see Listing 4.13).



My Blog

A simple blog ...

News on HTML

A preview of the new HTML elements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

Legal | Legal Notes | T&Cs | About me

Figure 4.17 The Header and Footer of the HTML Document Appear in Gray

```
...
  <body>
  <header>
    <hr />
    <h1>My Blog</h1>
    <p>A simple blog ...</p>
    <hr />
  </header>
  <h2>News on HTML</h2>
  <article>
    <h3>A preview of the new HTML elements</h3>
    <p>Lorem ipsum dolor ... </p>
  </article>
  <footer>
    <hr />
    <a href="#">Legal</a> |
    <a href="#">Legal Notes</a> |
    <a href="#">T&Cs</a> |
    <a href="#">About me</a>
    <hr />
  </footer>
  </body>
...
```

Listing 4.13 /examples/chapter004/4_2_7/index2.html

4.2.8 Using <main>: An HTML Element for the Main Content

I described the `div` element in the previous section, so it makes sense to deal with the `main element` at this point. Where `<div id="main">...</div>` was used in the past, you can use `<main>...</main>` from now on. The `id` attribute identifies an element that occurs only once within a document.

Like all other new HTML elements, you should use the `main` element as sensibly as possible. In practice, you use it for the main content of a website, which means it's best not to place it inside `<article>`, `<aside>`, `<footer>`, `<nav>`, or `header` elements.

In the web browser, the `main element` is rendered like the `div` element with no special properties and only creates a line break. However, unlike the `div` element, you should use the `main` element only once (visibly) in an HTML document. In contrast to the `<section>` element, the `main` element isn't a section element, but a pure grouping element. Thus, the use of such a section doesn't affect the heading structure (the document outline) of the HTML document.

Here's an example of how you can group a section as the main section of a web page:

```
...
  <body>
  <header>
    <h1>My Blog</h1>
    <p>A simple blog ...</p>
  </header>
  <main>
    <h2>News on HTML</h2>
    <article>
      <h3>A preview of the new HTML elements</h3>
      <p>Lorem ipsum dolor...</p>
    </article>
  </main>
  <footer>
    <a href="#">Legal</a> |
    <a href="#">Legal Notes</a> |
    <a href="#">T&Cs</a> |
    <a href="#">About me</a>
  </footer>
  </body>
...
```

Listing 4.14 /examples/chapter004/4_2_8/index.html

Using <main> Multiple Times?

`<main>` is intended to present the main content of an HTML document and should therefore be included only once in a document. If it's used more than once, then this page won't pass the validation check. Nevertheless, there are *single-page* web applications, that is, applications that consist of a single HTML document and whose content is dynamically reloaded, where this rule can become an issue. For this reason, the use of the `main` element has been adjusted somewhat, and multiple `main` elements can now be used. However, only one `<main>` element of those can be visible at a time. All other `main` elements must be provided with the `hidden` attribute. For example:

```
<main>...</main>
<main hidden>...</main>
<main hidden>...</main>
```

Although there are other ways in CSS to hide individual elements, you can use only the `hidden` attribute with `<main>` for the HTML document to be valid. All other options are invalid.

4.2.9 Labeling Content Separately Using <figure> and <figcaption>

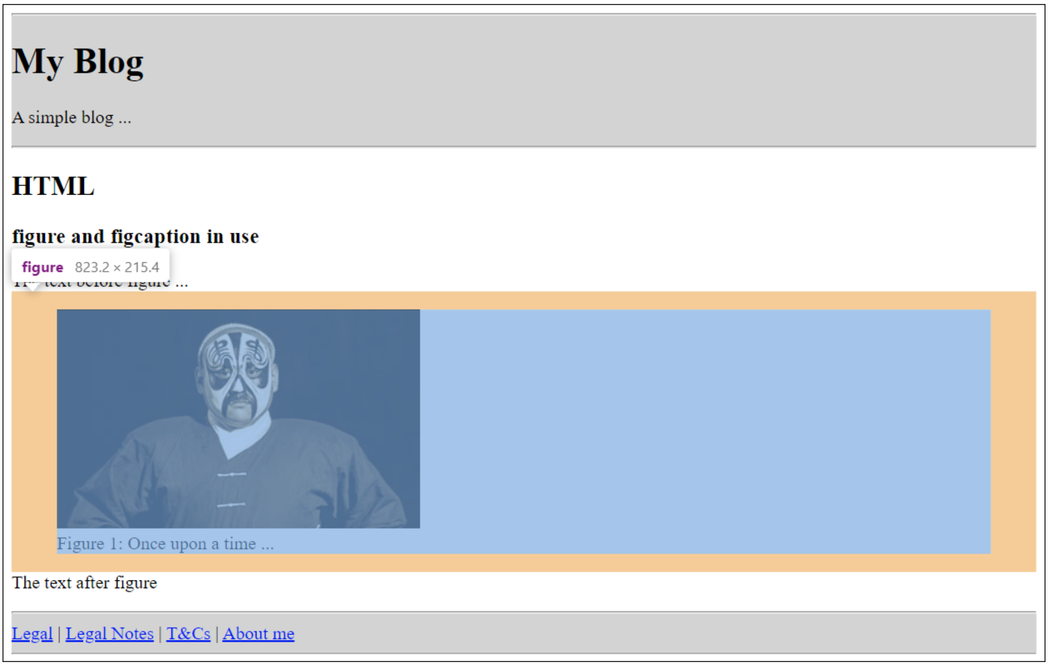
To set off or group certain content such as tables, images, listings, videos, or other HTML elements from the usual body text, you can use the `figure element`. If you want to link this section with an (optional) caption, you should use the `figcaption` element. Like the `figure` element, the `figcaption` element can contain other HTML elements besides ordinary body text. Thus, the `figure` element serves as the semantic parent for an element belonging to the page content, such as an image, table, listing, or other content, and the `figcaption` element encloses the subtitle to that element.

Here's a simple example, the result of which is shown in Figure 4.18:

```
...
  <h2>HTML</h2>
  <article>
    <h3>figure and figcaption in use</h3>
    <p>The text before figure ...</p>
    <figure>
      <img src="opera.jpg" alt="Chinese Opera" />
      <figcaption>Figure 1: Once upon a time ...</figcaption>
      </figure>
    <p>The text after figure</p>
  </article>
...
```

Listing 4.15 /examples/chapter004/4_2_9/index.html

If you want to place the (optional) caption with the `figcaption` element before the content (above the image in the example), you need to use the element right after the opening `<figure>`. However, it's only possible to use a `figcaption` element between `<figure>` and `</figure>`, and `<figcaption>` must be the first or last element of the `figure` element.


My Blog

A simple blog ...

HTML

figure and figcaption in use

The text before figure ...

Figure 1: Once upon a time ...

The text after figure

Legal | Legal Notes | T&Cs | About me

Figure 4.18 In the <article> Element between <figure> and </figure>, an Image Has Been Inserted with the <img> Element and a Caption with the <figcaption> Element

Between `<figure>` and `</figure>` you can also use more than one content type (e.g., an image in the example). In the web browser, a `figure` usually doesn't get displayed separately. In addition to a separate line, the content between `<figure>` and `</figure>` is often displayed slightly indented. However, CSS is used for the design of the `figure` element anyway.

4.2.10 Creating Unordered Lists Using <ul> and

An unordered list is basically nothing more than an unnumbered bulleted list in which all list entries are given a bullet character. The web browsers usually display this bullet with a *bullet point*.

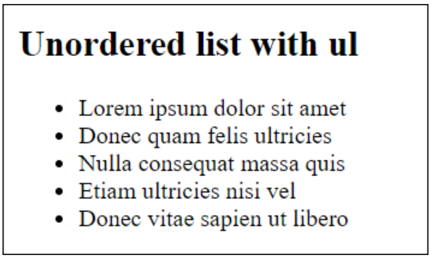
You can introduce such a list with an opening `<ul>` (`ul` = *unordered list*), followed by the actual bullet points, which you write between `<li>` and `</li>`. Each `li` element (`li` = *list item*) is a bullet point. At the end, you must end the unordered bullet list with the closing

`</ul>`. Only `li` elements can be contained between `<ul>` and `</ul>`. In between the `li` elements, you can also use other HTML elements (except for section elements).

Here's a simple example of an unordered list, the execution of which you can see in Figure 4.19.

```
...
<article>
  <h2>Unordered bullet list with ul</h2>
  <ul>
     <li>Lorem ipsum dolor sit amet</li>
     <li>Donec quam felis ultricies</li>
     <li>Nulla consequat massa quis</li>
     <li>Etiam ultricies nisi vel</li>
     <li>Donec vitae sapien ut libero</li>
  </ul>
</article>
...
```

Listing 4.16 /examples/chapter004/4_2_10_15/index.html



Unordered list with ul

- Lorem ipsum dolor sit amet
- Donec quam felis ultricies
- Nulla consequat massa quis
- Etiam ultricies nisi vel
- Donec vitae sapien ut libero

Figure 4.19 Bulleted Lists with the <ul> Element Are Usually Displayed with a Bullet Point

4.2.11 Creating Ordered Lists Using <ol> and

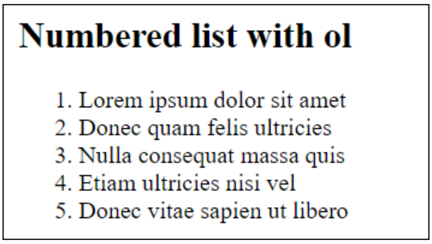
What you've just read about the `ul` element also applies to the `ol` element (`ol` = *ordered list*). The only exception is that the `ol` element is an ordered list—more precisely, a numbered list in which the individual `li` elements are automatically numbered.

Here's an example of an ordered list, the execution of which you can see in Figure 4.20.

```
...
<article>
  <h2>Numbered bullet list with ol</h2>
  <ol>
     <li>Lorem ipsum dolor sit amet</li>
     <li>Donec quam felis ultricies</li>
     <li>Nulla consequat massa quis</li>
```

```
    <li>Etiam ultricies nisi vel</li>
    <li>Donec vitae sapien ut libero</li>
  </ol>
</article>
...
```

Listing 4.17 /examples/chapter004/4_2_10_15/index.html



Numbered list with ol

1. Lorem ipsum dolor sit amet
2. Donec quam felis ultricies
3. Nulla consequat massa quis
4. Etiam ultricies nisi vel
5. Donec vitae sapien ut libero

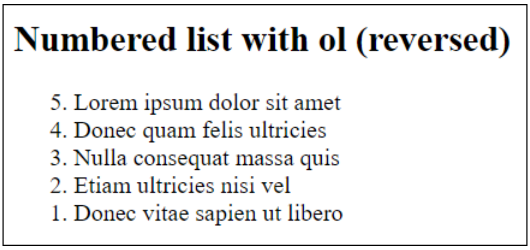
Figure 4.20 The Numbered List with the <ol> Element Uses Arabic Numerals by Default

4.2.12 Reversing the Numbering of an Ordered List

With HTML, it's also possible to reverse the order of numbering via the HTML attribute `reversed` in the opening `<ol>` tag, so that the numbering gets displayed in descending order. Based on the preceding example, you only need to insert the following:

```
<ol reversed="reversed">
  ...
</ol>
```

Besides `<ol reversed="reversed">`, you can also just use `<ol reversed>` here because it's a standalone attribute. But if you want to be XHTML compliant, you must use the form `<ol reversed="reversed">`. In HTML, you can use both versions.



Numbered list with ol (reversed)

5. Lorem ipsum dolor sit amet
4. Donec quam felis ultricies
3. Nulla consequat massa quis
2. Etiam ultricies nisi vel
1. Donec vitae sapien ut libero

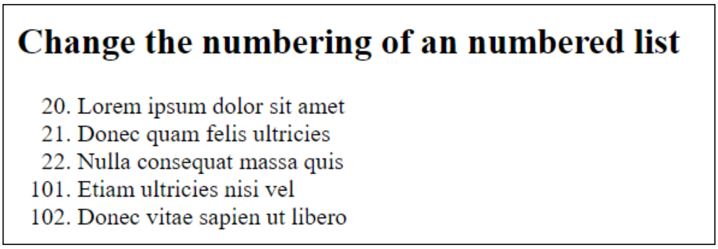
Figure 4.21 The Numbering Order Was Reversed via the "reversed" Attribute

4.2.13 Changing the Numbering of an Ordered List

You can use the HTML attribute `start` to specify the start value of the first `li` element in the opening `<ol>` tag. All values that follow the first `li` element are incremented by the value `1`. Even within an opening `<li>` tag, you can use the HTML attribute `value` to change the numbering of the list entry. All subsequent entries are incremented by the value `1` using the value specified in `value`.

The execution of the following example is shown in Figure 4.22.

```
...
<ol start="20">
  <li>Lorem ipsum dolor sit amet</li>
  <li>Donec quam felis ultricies</li>
  <li>Nulla consequat massa quis</li>
  <li value="101">Etiam ultricies nisi vel</li>
  <li>Donec vitae sapien ut libero</li>
</ol>
...
```



Change the numbering of an numbered list

20. Lorem ipsum dolor sit amet
21. Donec quam felis ultricies
22. Nulla consequat massa quis
101. Etiam ultricies nisi vel
102. Donec vitae sapien ut libero

Figure 4.22 The Starting Numbering Was Set to 20 Right in the Opening <ol> Tag with the Attribute “start” and Then Again in an Opening <li> Tag with the Attribute “value” to 101

4.2.14 Nesting Lists within Each Other

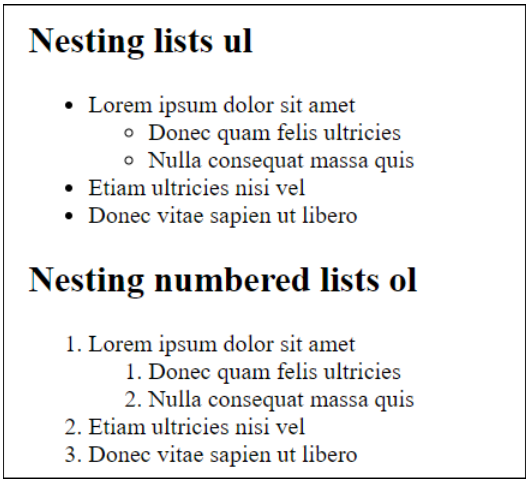
You can nest both numbered lists and bulleted lists. Such nested lists are used when, for example, you need a finer structuring of the lists, such as a table of contents (e.g., at the beginning of this book). A navigation with submenus is also often formulated by means of a bulleted list.

Nesting lists can get a little messy, so if you have a deeper bulleted hierarchy, you should use indentations and/or add a comment. When nesting, the `li` elements aren’t nested inside each other, as might be assumed, but a `ul` or `ol` element must be written again with the nested `li` elements inside an opening parent `<li>` tag. Only when you close the opening `<li>` tag with `</li>` will this list be marked and displayed as a child list.

The execution of the following code snippet is shown in Figure 4.23.

```
...
<h2>Nesting bullet lists ul</h2>
<ul>
  <li>Lorem ipsum dolor sit amet
  <ul><!-- Start: 1. Nesting -->
    <li>Donec quam felis ultricies</li>
    <li>Nulla consequat massa quis</li>
  </ul><!-- End: 1. Nesting -->
  </li>
  <li>Etiam ultricies nisi vel</li>
  <li>Donec vitae sapien ut libero</li>
</ul>
...
<h2>Nesting numbered lists ol</h2>
<ol>
  <li>Lorem ipsum dolor sit amet
  <ol><!-- Start: 1. Nesting -->
    <li>Donec quam felis ultricies</li>
    <li>Nulla consequat massa quis</li>
  </ol><!-- End: 1. Nesting -->
  </li>
  <li>Etiam ultricies nisi vel</li>
  <li>Donec vitae sapien ut libero</li>
</ol>
...
```

Listing 4.18 /examples/chapter004/4_2_10_15/index.html



Nesting lists ul

- Lorem ipsum dolor sit amet
  - Donec quam felis ultricies
  - Nulla consequat massa quis
- Etiam ultricies nisi vel
- Donec vitae sapien ut libero

Nesting numbered lists ol

1. Lorem ipsum dolor sit amet
   1. Donec quam felis ultricies
   2. Nulla consequat massa quis
2. Etiam ultricies nisi vel
3. Donec vitae sapien ut libero

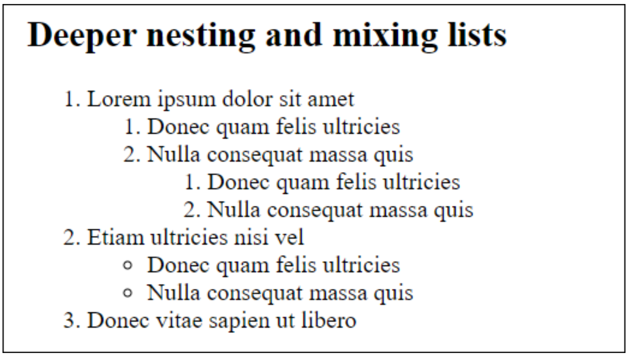
Figure 4.23 The Nesting of Unnumbered Lists and Numbered Lists during Execution

Of course, you can nest the lists even deeper. Mixing unordered and ordered lists is also possible without any problem. Unfortunately, it isn't possible to force automatic numbering such as 1.2, 1.3, 1.4, and so on for the numbered sublists.

To illustrate this, the following is an example of a deeper and mixed nesting, the result of which is shown in Figure 4.24.

```
...
<h2>Deeper nesting and mixing lists</h2>
<ol>
  <li>Lorem ipsum dolor sit amet
  <ol><!-- Start: 1. Nesting -->
      <li>Donec quam felis ultricies</li>
      <li>Nulla consequat massa quis
      <ol><!-- Start: 2. Nesting -->
          <li>Donec quam felis ultricies</li>
          <li>Nulla consequat massa quis</li>
      </ol><!-- End: 2. Nesting -->
      </li>
  </ol><!-- End: 1. Nesting -->
  </li>
  <li>Etiam ultricies nisi vel
  <ul><!-- Start: 1. Nesting (bullet point) -->
      <li>Donec quam felis ultricies</li>
      <li>Nulla consequat massa quis</li>
  </ul><!-- End: 1. Nesting (bullet point) -->
  </li>
  <li>Donec vitae sapien ut libero</li>
</ol>
...
```

Listing 4.19 /examples/chapter004/4_2_10_15/index.html



Deeper nesting and mixing lists

1. Lorem ipsum dolor sit amet
    1. Donec quam felis ultricies
    2. Nulla consequat massa quis
        1. Donec quam felis ultricies
        2. Nulla consequat massa quis
2. Etiam ultricies nisi vel
    - Donec quam felis ultricies
    - Nulla consequat massa quis
3. Donec vitae sapien ut libero

Figure 4.24 Further Nesting Depths and Mixing of Ordered and Unordered Lists

Omitting the Closing Tag from Lists

As you may remember from Chapter 2, Section 2.1.6, it's possible to omit the closing tags in some places. Especially if a list is deeply and extensively nested, this may even be clearer and easier than setting the closing tags. As mentioned earlier, this style isn't used in this book, and I've never used it in practice (yet). Nevertheless, it should be pointed out here because the lists are listed as a pro-argument, especially by the "omission faction."

4.2.15 Creating a Description List Using <dl>, <dt>, and <dd>

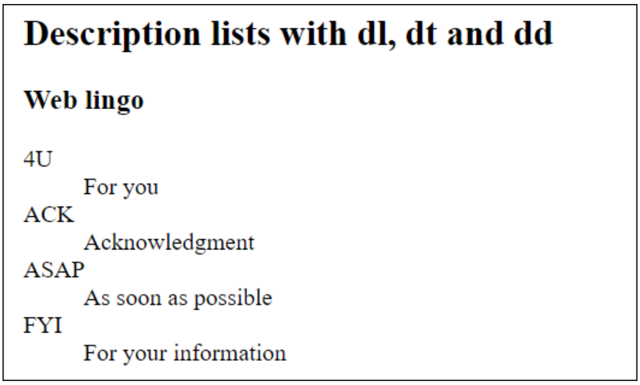
In HTML, there's another type of list you can use—the description list. This list is more of a name-value mapping list. Typical use cases for the description list are glossaries or the listing of special metadata and values; in other words, it's simply a special list with certain data in which a value or a description is assigned.

A description list gets summarized between `<dl>` and `</dl>` (`dl` = *description list*). The `dl` element may only contain the `dt` and `dd` elements described in the same way. The expression to be described, that is, the name of the name-value mapping list, is marked with `<dt>` and `</dt>` (`dt` = *description term*). The associated description is written after the `dt` element between `<dd>` and `</dd>` (`dd` = *definition description*). In turn, other HTML elements may be used in `dt` and `dd` elements—except for *grouping* elements and HTML elements for new sections (*sectioning*).

Here's a simple example of a description list, the result of which is shown in Figure 4.25. By default, web browsers display the descriptions (`<dd>` elements) slightly indented compared to the expression (`<dt>` elements). Here the description list was used for a list of abbreviations in the web jargon.

```
...
<h3>Web lingo</h3>
<dl>
  <dt>4U</dt>
  <dd>For you</dd>
  <dt>ACK</dt>
  <dd>Acknowledgment</dd>
  <dt>ASAP</dt>
  <dd>As soon as possible</dd>
  <dt>FYI</dt>
  <dd>For your information</dd>
</dl>
...
```

Listing 4.20 /examples/chapter004/4_2_10_15/index.html



Description lists with dl, dt and dd

Web lingo

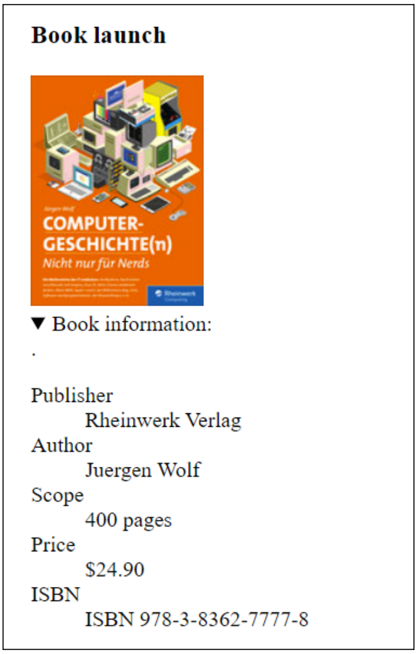
4U
For you
ACK
Acknowledgment
ASAP
As soon as possible
FYI
For your information

Figure 4.25 Descriptions (<dd> Elements) Slightly Indented Compared to the Expression (<dt> Elements)

Such a name-value pair list can also be used within other HTML elements such as between `<aside>` and `</aside>` or the new `details` element, as shown in the following example:

```
...
<h3>Book launch</h3>
<img src="Computer stories.png"
alt="Computer stories"><br>
<details>
  <summary>Book information:</summary>.
  <dl>
    <dt>Publisher</dt>
    <dd>Rheinwerk Verlag</dd>
    <dt>Author</dt>
    <dd>Juergen Wolf</dd>
    <dt>Scope</dt>
    <dd>400 pages</dd>
    <dt>Price</dt>
    <dd>$24.90</dd>
    <dt>ISBN</dt>
    <dd> ISBN 978-3-8362-7777-8 </dd>
  </dl>
</details>
...
```

Listing 4.21 /examples/chapter004/4_2_10_15/index.html



Book launch

▼ Book information:

.

Publisher
    Rheinwerk Verlag
Author
    Juergen Wolf
Scope
    400 pages
Price
    $24.90
ISBN
    ISBN 978-3-8362-7777-8

Figure 4.26 The Description List for an Image (a Book) Has Been Wrapped inside the <details> Element, Allowing the Description to be Expanded and Collapsed

4.3 Using Semantic HTML

Now that you know the HTML elements for page structuring and text structuring, you may be wondering how you can create a basic web page with these HTML elements so that it makes sense semantically. Specifically, this means that you can define the different logical parts of a web page with HTML tags. By the way, the semantic web isn't just a fad, but helps search engines, for example, better allocate the sheer flood of data on the internet. Search engines such as Google even prefer semantic web pages and searches HTML pages for semantic content.

Let's take as a simple example the term *goal*, whose meaning in hockey is different from that in business. The term gets its assignment and meaning only if you provide the relevant context. This is roughly how you can imagine the semantic web: you contextualize the content with code so that machines can also interpret and process it.

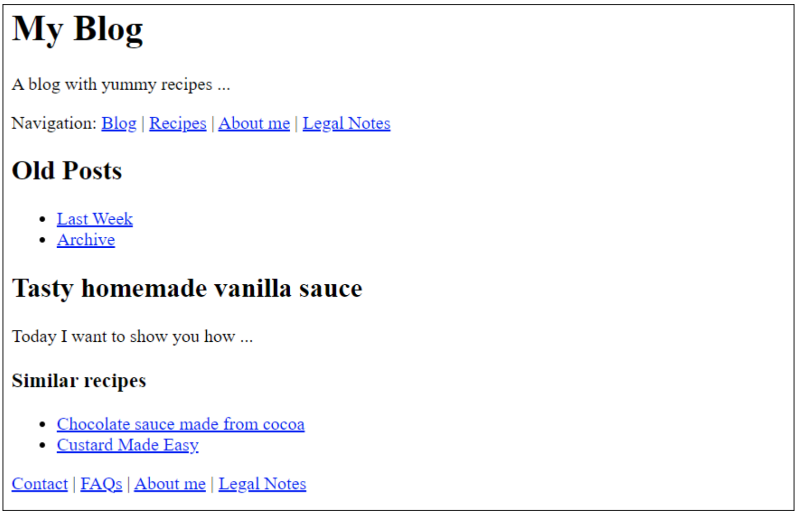
4.3.1 HTML without a Precise Structure

The first example is a classic HTML document that has no detailed structure:

```
...
  <h1>My Blog</h1>
  <p>A blog with yummy recipes ...</p>
  <p>Navigation:
    <a href="#">Blog</a> | <a href="#">Recipes</a> |
    <a href="#">About me</a> | <a href="#">Legal Notes</a>
  </p>
  <h2>Old Posts</h2>
  <ul>
    <li><a href="#">Last Week</a></li>
    <li><a href="#">Archive</a></li>
  </ul>
  <h2>Tasty homemade vanilla sauce</h2>
  <p>Today I want to show you how ...</p>
  <h3>Similar recipes</h3>
  <ul>
     <li><a href="#">Chocolate sauce made from cocoa</a></li>.
     <li><a href="#">Custard Made Easy</a></li>
  </ul>
  <p>
    <a href="#">Contact</a> | <a href="#">FAQs</a> |
    <a href="#">About me</a> | <a href="#">Legal Notes</a>
  </p>
...
```

Listing 4.22 /examples/chapter004/4_3_1/index.html

There isn't much to note about this example. The HTML code is valid and can be used like that. You can see some headings, unordered lists, navigation, and various paragraph texts. However, such code is rarely used because it's nearly unstructured and is relatively poorly suited for styling or laying out via CSS.



My Blog

A blog with yummy recipes ...

Navigation: Blog | Recipes | About me | Legal Notes

Old Posts

- Last Week
- Archive

Tasty homemade vanilla sauce

Today I want to show you how ...

Similar recipes

- Chocolate sauce made from cocoa
- Custard Made Easy

Contact | FAQs | About me | Legal Notes

Figure 4.27 /examples/chapter004/4_3_1/index.html When Displayed in the Web Browser

4.3.2 Generic Structuring Using <div>

The first example didn't contain any element to tell you where the different sections of content were located. For this reason, we'll now use the `div` element to divide the content into separate sections. Take a look at the same example again now, but this time it contains the `div` elements:

```
...
  <div>
  <h1>My Blog</h1>
  <p>A blog with yummy recipes ...</p>
  </div>
  <div>
  <p>Navigation:
    <a href="#">Blog</a> |
    <a href="#">Recipes</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </p>
  </div>
  <div>
  <h2>Old Posts</h2>
  <ul>
    <li><a href="#">Last Week</a></li>
    <li><a href="#">Archive</a></li>
  </ul>
  </div>
  <div>
  <h2>Tasty homemade vanilla sauce</h2>
  <p>Today I want to show you ... </p>
  <h3>Similar recipes</h3>
  <ul>
     <li><a href="#">Chocolate sauce made from cocoa</a></li>.
     <li><a href="#">Custard Made Easy</a></li>
  </ul>
  </div>
  <div>
  <p>
    <a href="#">Contact</a> |
    <a href="#">FAQs</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </p>
  </div>
...
```

Listing 4.23 /examples/chapter004/4_3_2/index.html

This time, the content was separated by means of `div` elements. Nevertheless, we still don't see any semantic elements. Visually, nothing changes here compared to the */examples/chapter004/4_3_1/index.html* example from the previous section.

All that can be achieved by using the `div` element is to group a piece of content together. Thus, it depends on the author of the web page to assign meaning to the individual `div` elements. Before semantic elements came into play, this was done via attributes in the opening `<div>` tag. So, let's now take a look at the next step and the next example, in which the individual `div` elements get their meaning:

```
...
  <div id="header">
  <h1>My Blog</h1>
  <p>A blog with yummy recipes ...</p>
  </div>
  <div id="navigation">
  <p>Navigation:
    <a href="#">Blog</a> |
    <a href="#">Recipes</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </p>
  </div>
  <div id="sidebar">
  <h2>Old Posts</h2>
  <ul>
    <li><a href="#">Last Week</a></li>
    <li><a href="#">Archive</a></li>
  </ul>
  </div>
  <div id="content">
  <h2>Tasty homemade vanilla sauce</h2>
  <p>Today I want to show you ...</p>
  <h3>Similar recipes</h3>
  <ul>
     <li><a href="#">Chocolate sauce made from cocoa</a></li>.
     <li><a href="#">Custard Made Easy</a></li>
  </ul>
  </div>
  <div id="footer">
  <p>
    <a href="#">Contact</a> |
    <a href="#">FAQs</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </p>
  </div>
...
```

Listing 4.24 /examples/chapter004/4_3_2/index2.html

While nothing has changed in a purely visual sense, the `div` elements have gained meaning thanks to the `id` attribute. We now have a header, navigation, sidebar, content, and footer as it's visually represented in Figure 4.28. Using CSS, you can design and lay out these areas individually. This way, you can virtually already achieve a semantically correct structuring of the website, but not yet a semantically unified structuring.

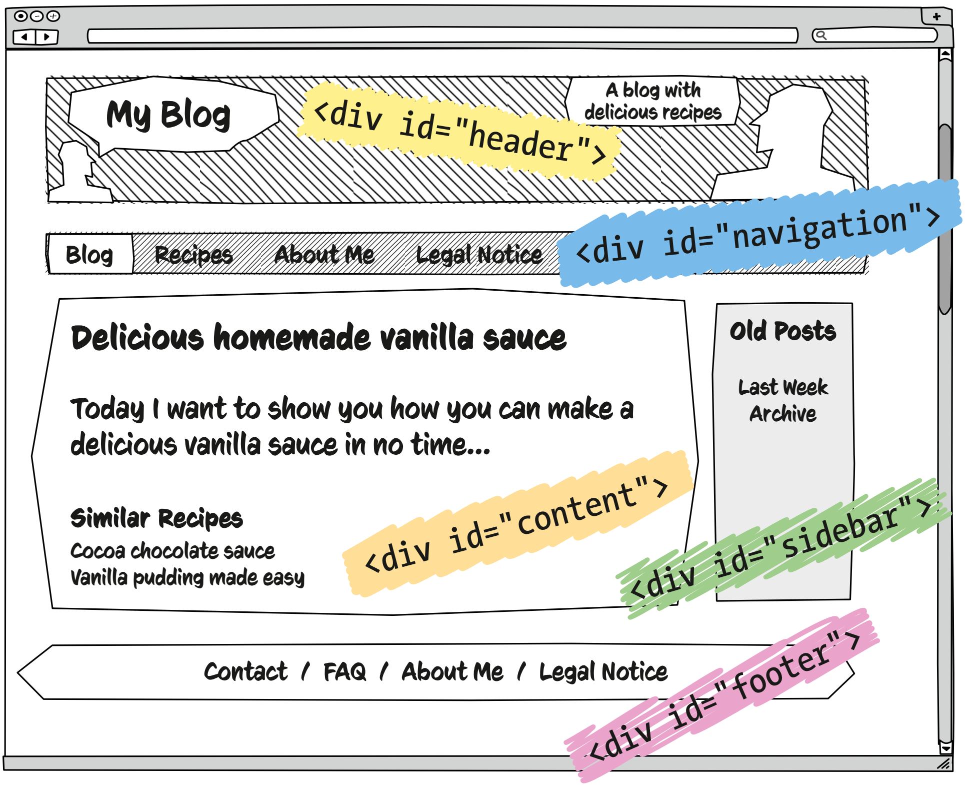

My Blog
<div id="header">
A blog with delicious recipes
Blog
Recipes
About Me
Legal Notice
<div id="navigation">
Delicious homemade vanilla sauce
Today I want to show you how you can make a delicious vanilla sauce in no time...
Similar Recipes
Cocoa chocolate sauce
Vanilla pudding made easy
<div id="content">
Old Posts
Last Week
Archive
<div id="sidebar">
Contact / FAQ / About Me / Legal Notice
<div id="footer">

Figure 4.28 The Meaning for the Layout Areas Is Assigned via <div> and the "id" Attribute

So why should you use semantic structuring at all when you can work with `div` elements without any problem? There are several reasons for this: Despite the use of IDs in the `div` element, you have a semantically neutral element. It isn't a standardized structuring, but instead everyone can define what they want. For the machines, there's still no difference here. They can't know what you really mean by `id="header"` or `id="content"` and what's behind it. You might as well write `id="head"` or `id="synopsis"` or whatever in all the languages of the world.

For example, imagine a smart screen reader reading the main content of the web page to a visually impaired person. How would the screen reader know what the main content is? One web developer may write `id="content"`, another may write `id="main"`, and you may write `id="musings"`. In addition, some web developers don't mark up the main content at all.

The situation is the same with search engines. For search engines to return a better result, it's helpful if they know what belongs to the main content of the web page. Again, the search engine faces nonstandard class and ID names. Thus, it's an advantage here if you tell the web crawler on the next visit: this is the main content of my site.

<div> Can Still Be Used in HTML

Using `div` elements and labeling the layout sections with the ID and class names are by no means incorrect—they represent valid HTML. In addition, the `div` element often helps you solve a problem. Nevertheless, for future projects, you should use the new semantically meaningful elements that were introduced especially for this purpose.

4.3.3 Semantic Structuring Using the Elements Provided in HTML

To write a semantically meaningful structure as HTML code for machines, there are suitable elements in HTML that have already been described in the book and are listed once again in Table 4.3.

HTML Element	Meaning	Section in This Chapter
`header`	Header sections	Section 4.1.4
`nav`	Navigation blocks	Section 4.1.2 and "Declaring Content as a Page Navigation Bar Using <nav>" on Page 102
`section`	Division into content sections	"Dividing Content into Topic-Based Sections Using <section>" on Page 97
`article`	Division into self-contained blocks	"Dividing Content into a Self-Contained Block Using <article>" on Page 98
`aside`	Additional Information	Section 4.1.2
`footer`	Footer sections	Section 4.1.4

Table 4.3 Semantic HTML Elements

Returning to our example */examples/chapter004/4_3_2/index2.html*, this HTML code should do without `div` elements and instead rely on semantic HTML elements:

```
...
  <header>
  <h1>My Blog</h1>
  <p>A blog with yummy recipes ...</p>
  </header>
  <nav>
  <p>Navigation:
    <a href="#">Blog</a> |
    <a href="#">Recipes</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </p>
  </nav>
  <aside>
  <h2>Old Posts</h2>
  <ul>
    <li><a href="#">Last Week</a></li>
    <li><a href="#">Archive</a></li>
  </ul>
  </aside>
  <article>
  <h2>Tasty homemade vanilla sauce</h2>
  <p>Today I want to show you ...</p>
  <h3>Similar recipes</h3>
  <ul>
     <li><a href="#">Chocolate sauce made from cocoa</a></li>.
     <li><a href="#">Custard Made Easy</a></li>
  </ul>
  </article>
  <footer>
  <p>
    <a href="#">Contact</a> |
    <a href="#">FAQs</a> |
    <a href="#">About me</a> |
    <a href="#">Legal Notes</a>
  </p>
  </footer>
...
```

Listing 4.25 /examples/chapter004/4_3_3/index.html

When you look at the HTML document with the semantic elements, it's probably already much easier to recognize at first glance what has which meaning here. This is just a simple example. Here you can immediately see the header, navigation, sidebar, main content, and footer (see Figure 4.29). This way, the content could perhaps also be placed inside the `main` element in which the individual articles are then summarized using the `article` element.

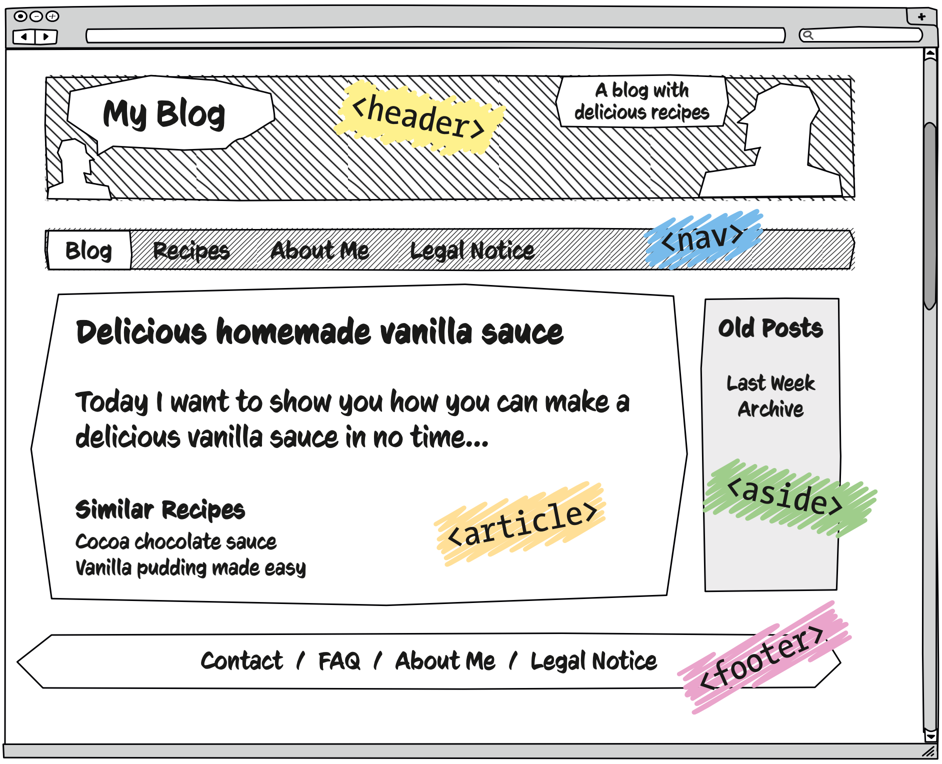

My Blog
<header>
A blog with delicious recipes
Blog
Recipes
About Me
Legal Notice
<nav>
Delicious homemade vanilla sauce
Today I want to show you how you can make a delicious vanilla sauce in no time...
Similar Recipes
Cocoa chocolate sauce
Vanilla pudding made easy
<article>
Old Posts
Last Week
Archive
<aside>
Contact / FAQ / About Me / Legal Notice
<footer>

Figure 4.29 Layout Areas Marked with HTML Semantic Elements

The point here isn't at all where you can use exactly which HTML element in detail, but rather that this semantic structuring makes sense, even if you've never heard of new elements such as `nav`, `article`, `header`, `footer`, and so on. The logic here is almost self-evident. It's much easier to see where the navigation, header, or footer is written in this document.

The Main Content with <main>

If you still want to summarize the main content of the web page in `<main>` in the example */examples/chapter004/4_3_3/index.html*, then you should choose the `article` element.

4.3.4 What's the Use of Those Semantic HTML Elements?

If you've carefully followed this section, you've seen that the semantic HTML elements are very useful. Thus, probably one of their advantages is that they make life easier for you as the developer of the website.

For "normal" visitors, these semantic HTML elements don't have much value at first. On the contrary, those visitors won't even be able to distinguish in the web browser whether you've used `div` elements or the semantic HTML elements. However, if, for example, a new smart web browser provides special features that let you get to navigation by clicking a button, the new semantics take on meaning for normal visitors as well.

The situation is different, however, for visually impaired visitors who use a screen reader. A good screen reader could "recognize" the content of the web page based on the new semantic structure and thus jump directly to the content or navigation.

Of course, you shouldn't disregard the search engines at all. For example, you could let the search engine know in a consistent and standardized way where which content is located, so that it assigns a higher ranking to the relevant content of a web page.

4.4 HTML Elements for Text Markups

You apply HTML elements for text markup within plain text for individual letters, words, or parts of sentences. Thus, the described elements don't create a new paragraph or line break, but mark out specific passages in a continuous text according to the semantics defined for the element. You can find all text markups used here in the HTML document */examples/chapter004/4_4/index.html*.

Text Formatting via CSS

Even though many of the elements presented here cause a slight visual change of the text in the web browser, you shouldn't use these HTML elements for text formatting. CSS is responsible for text formatting. These HTML elements rather serve a clean semantic text markup. Thanks to semantic text markup, you can lay the foundation for later text formatting with CSS. If you use sensible text markup in body text, you can later format your text more easily and logically with CSS.

HTML Element	Meaning
`<abbr>`	Marking abbreviations or acronyms.
`<cite>`	Marking text as source text of a working title.

Table 4.4 Brief Overview of the Elements Covered for Text Markups

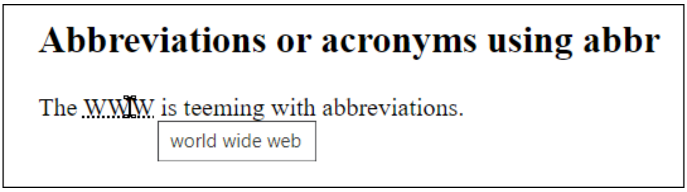
HTML Element	Meaning
`<code>`	Marking up computer code within a paragraph of text.
`<pre>`	Marking up preformatted text. All spaces and line breaks get displayed as specified in the text.
`<kbd>`	Marking up text as keyboard input.
`<samp>`	Marking up text as screen output of a program.
`<dfn>`	Defining a term.
`<var>`	Marking up text as a variable.
`<bdo>`	Changing the text direction for bidirectional text.
`<bdi>`	Defining a section for bidirectional text.
`<em>`	Highlighting text you would emphasize in spoken language.
`<strong>`	Highlighting words or passages that are particularly important in terms of content.
`<i>`	Marking up words or passages with technical terms, thoughts, and foreign words.
`<b>`	Marking up meaningful names or keywords.
`<mark>`	Highlighting text with a marker.
`<q>`	Marking up words or passages as cited or spoken text.
`<u>`	Marking up text underlined as proper name or incorrect words or passages.
`<s>`	Marking up text as no longer valid or obsolete.
`<ins>`	Marking up text as newly added in the revised sense.
`<del>`	Marking up text as deleted in the revised sense.
`<sub>`	Marking text as subscript.
`<sup>`	Marking text as superscript.
`<time>`	Marking up dates and times
`<small>`	Marking up text as small print, such as for copyright information, licensing information, or legal notes.
`<ruby>`, `<rp>`, and `<rt>`	Specifying Ruby annotations.
`<span>`	Marking up a general section within a paragraph of text.

Table 4.4 Brief Overview of the Elements Covered for Text Markups (Cont.)

4.4.1 Marking Up Abbreviations or Acronyms Using <abbr>

The `abbr` element (`abbr` = *abbreviation*) can be used for abbreviations or acronyms. It's also helpful to use the global HTML attribute `title` in which you write out the abbreviation or acronym so that a web browser can display the full meaning when hovering over it, as you can see in Figure 4.30. This reproduces the code snippet of the following example:

```
...
<p>The <abbr title="world wide web">WWW</abbr> is teeming with
  abbreviations.
</p>
...
```



Abbreviations or acronyms using abbr

The WWW is teeming with abbreviations.

world wide web

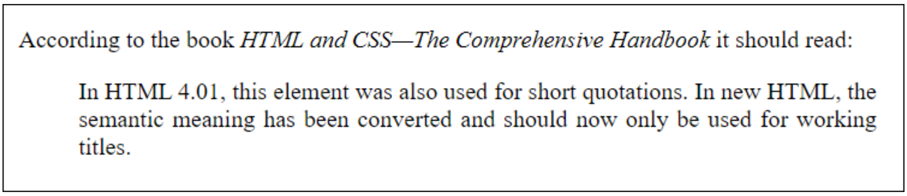
Figure 4.30 The Global "title" Attribute Displays the Meaning of the Abbreviation "WWW" When You Hover the Mouse Cursor over the Word

Anyone who writes abbreviations between `<abbr>` and `</abbr>` is passing useful information to the web browser, language-checking software, translation systems, screen readers, or even the search engines for indexing. The extent to which this information is useful and actually used can't always be predicted. Nevertheless, the `abbr` element is very useful for logical text markup.

4.4.2 Marking Up Text as the Source of a Working Title Using <cite>

You can use the `cite` element when you include the title of a book, movie, painting, piece of music, exhibition, and so on in the body text. However, you should only mark up the working title and not the name or main character of the title. Again, you can usually still use the global HTML attribute `title` to specify more information about the working title when the user hovers over it with the mouse pointer. Most web browsers display everything between `<cite>` and `</cite>` in italics.

```
...
<p>According to the book <cite>HTML and CSS—The Comprehensive
  Handbook</cite> it should read:
</p>
...
```



According to the book *HTML and CSS—The Comprehensive Handbook* it should read:

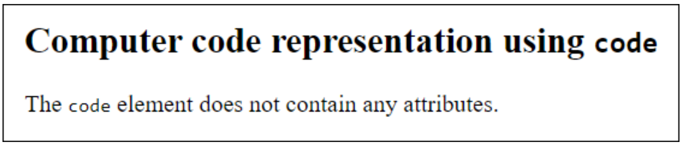
> In HTML 4.01, this element was also used for short quotations. In new HTML, the semantic meaning has been converted and should now only be used for working titles.

Figure 4.31 In This Example, We Wrote the Working Title of a Book between <cite> and </cite>

4.4.3 Marking Up Computer Code Representation Using <code> and <pre>

You should use the `code` element to indicate computer code within body text. Most web browsers often display this area using a monospace font such as Courier, as shown in Figure 4.32, which renders the following code snippet in the web browser:

```
...
<h2>Computer code with <code>code</code></h2>
<p>The <code>code</code> element does not contain any attributes.</p>
...
```



Computer code representation using `code`

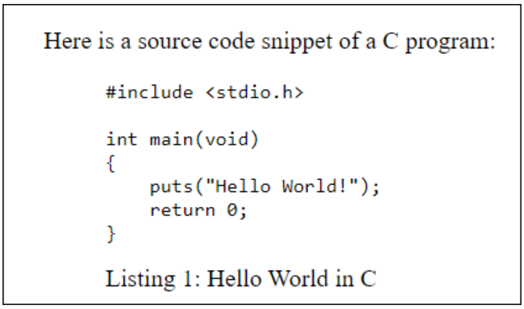
The `code` element does not contain any attributes.

Figure 4.32 The <code> Element Is Suitable for Marking Up Language Elements or Parts of a Source Code of a Particular Language

If you want to format multiple lines of computer code, you should note the `code` elements in between `<pre>` and `</pre>`. The `pre` element (`pre` = *preformatted*) represents preformatted text. In the section between `<pre>` and `</pre>`, several whitespace characters won't get combined to one space, but everything is output as it was entered in the editor. Because that section is output in a monospace font, the `pre` element is very suitable for outputting source text across multiple lines. This isn't to say that `<pre>` is only suitable for marking up source code. It's therefore recommended that you specify the content between `<pre>` and `</pre>` more precisely with appropriate text markup. So, for source `code`, you should use `<code>`; for keyboard input, `<kbd>`; and for displaying a program output, `<samp>`.

In the following example, the text preformatted between `<pre>` and `</pre>` is output as it was written. Specifying `<code>` and `</code>` between `<pre>` and `</pre>` isn't mandatory, but it makes the text markup even more precise. You can see the following example at execution in Figure 4.33:

```
...
<p>Here is a source code snippet of a C program:</p>
<pre><code>#include &lt;stdio.h&gt;

int main(void)
{
    puts("Hello World!");
    return 0;
}</code></pre>
...
```



Here is a source code snippet of a C program:

```
#include <stdio.h>

int main(void)
{
    puts("Hello World!");
    return 0;
}
```

Listing 1: Hello World in C

Figure 4.33 The Text Preformatted between <pre> and </pre> Gets Output Exactly as It Was Entered

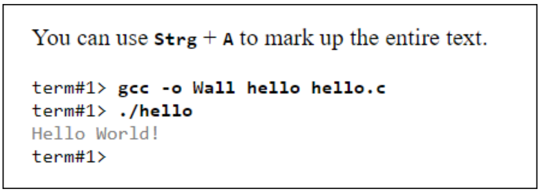
The Masking Characters "&lt;" and "&gt;"

To display < or > characters in HTML that aren't to be used as HTML, the character entities `&lt;` for < and `&gt;` for > were used here.

4.4.4 Keyboard Input Using <kbd> and Program Output Using <samp>

The `kbd` element (`kbd` = *keyboard*) should be used to mark up continuous text as keyboard input. The `samp` element, on the other hand, should be used for the screen output of programs. Most often, these two elements are also rendered in a monospace font (usually Courier) in the web browser, as you can see in Figure 4.34, which shows the following example running in the web browser:

```
...
<p>You can use <kbd>Strg</kbd> + <kbd>A</kbd> to mark up the entire text.</p>
<pre>term#1&gt; <kbd>gcc -o Wall hello hello.c</kbd>
term#1&gt; <kbd>./hello</kbd>
<samp>Hello World!</samp>
term#1&gt;</pre>
...
```



You can use **Strg** + **A** to mark up the entire text.

```
term#1> gcc -o Wall hello hello.c
term#1> ./hello
Hello World!
term#1>
```

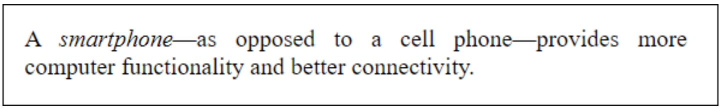
Figure 4.34 The Web Browsers Themselves Decide How to Display the Text Between <kbd> and <kbd> for Input or <samp> and </samp>

In Figure 4.34, the `kbd` elements have been made bold, and the `samp` elements have been made gray by using CSS to help you see what has been used where.

4.4.5 Marking Up Text as a Definition Using <dfn>

Text that you write between `<dfn>` and `</dfn>` is supposed to represent a definition. Usually, you mark up a word or a text passage, which you then explain in the text that follows. However, the `dfn` element shouldn't mark up the definition itself, but the defined term. Let's take a look at a simple example:

```
...
<p>A <dfn>smartphone</dfn>–as opposed to a
   cell phone–provides more
   computer functionality and better connectivity.
</p>
...
```



A *smartphone*—as opposed to a cell phone—provides more computer functionality and better connectivity.

Figure 4.35 In this Paragraph Text, the Term "Smartphone" Was Described, Which Is Why It Was Placed between <dfn> and </dfn>

In common practice, you can also use another element such as `<abbr>` inside `<dfn>` and `</dfn>`, as shown in the following example:

```
...
<p>A <dfn><abbr>smartphone<abbr></dfn>–as opposed to
   a cell phone–provides more
   computer functionality and better connectivity.
</p>
...
```

You can also use the global attribute `title` inside the opening `<dfn>` tag. The value of `title` should be the same as the content of the `dfn` element.

4.4.6 Marking Up Text as a Variable Using <var>

You can use the `var` element to mark up the text in between as a variable. Such a variable can be, for example, part of an application, a mathematical expression, or an identifier of a variable in a programming language:

```
...
<p>The radius <var>r</var> is equal to
   half the diameter <var>d</var>.
</p>
...
```

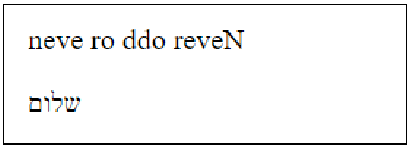
4.4.7 Changing the Text Direction Using <bdo> and <bdi>

The `bdo` element (`bdo` = *bidirectional override*) allows you to change the text direction. This is useful, for example, when you want to display text that is written from right to left (e.g., Hebrew or Arabic). By default, the text is displayed from left to right. To change the text direction, you must use the global HTML attribute `dir`. The attribute value `rtl` makes the text run from right to left, whereas `ltr` makes it run from left to right.

You don't need to put every Hebrew or Arabic word between `<bdo dir="rtl">` and `</bdo>`. When you use Unicode in HTML, the text direction is usually automatically taken into account according to the language, provided you use a Unicode-capable web browser. You should only use the `bdo` element if the correct text direction doesn't work.

To illustrate this, here's a code snippet in which in the first paragraph text—a palindrome for fun—was put between `<bdo>` and `</bdo>`, and the text alignment was changed via the attribute `dir` into the value `rtl` (*right to left*). The second paragraph, on the other hand, displays the Hebrew word "shalom", which usually doesn't require changing the text direction. You can see the result of these lines in Figure 4.36:

```
<p><bdo dir="rtl">Never odd or even</bdo></p>
<p>שלום</p>
```



neve ro ddo reveN

שלום

Figure 4.36 Example Executed with <bdo>

The situation is different in the following HTML lines:

```
<p>1: السلام عليكم(as-salaam alaykum)</p>
<p>2: שלום(shalom)</p>
<p>Howdy: 3</p>
```

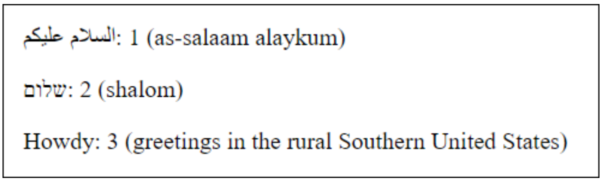
The first two examples would probably not lead to the desired result here. Although `as-salaam alaykum` in Arabic and `shalom` in Hebrew are correctly written from right to left, the following colon and number have been given a different writing direction. Originally, this was supposed to look like the third paragraph with `Howdy`.

Here, the writing behind the Arabic characters continued from right to left, so that the colon and the number behind it also retained the writing direction. Only the translation of the Arabic or Hebrew meaning was reproduced in the correct place.

To be on the safe side, the `bdi` element (`bdi` = *bidirectional isolation*) was introduced for this purpose. Using the `bdi` element, you can mark up the boundaries of text direction changes in a Unicode-enabled web browser more accurately. Thus, in the preceding example, you only need to put the Arabic or Hebrew characters between `<bdi>` and `</bdi>`.

After that, it looks as shown in Figure 4.37:

```
<p><bdi>السلام عليكم</bdi>: 1 (as-salaam alaykum)</p>
<p><bdi>שלום</bdi>: 2 (shalom)</p>
<p>Howdy: 3</p>
```



السلام عليكم: 1 (as-salaam alaykum)

שלום: 2 (shalom)

Howdy: 3 (greetings in the rural Southern United States)

Figure 4.37 Thanks to the Containment of the Arabic and Hebrew Script between <bdi> and </bdi>, the Colon and the Decimal Number Now Display after the Script

4.4.8 Emphasizing Text Using <em>, <strong>, <i>, and <b>

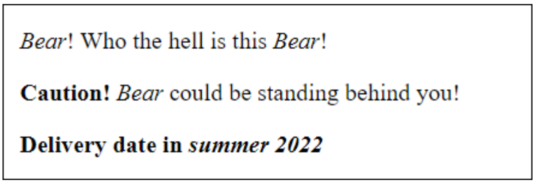
To emphasize text, you can use either the `em` element (`em` = *emphasis*) or the `strong` element. The `em` element should be used for words or passages that you would emphasize when speaking.

If you want to bring a word or passage more into focus, you should use the `strong` element. In contrast to the `em` element, the `strong` element is used to mark certain places in the text with a special signal or warning effect. The `strong` element should definitely be used for words or passages that are particularly important in terms of content.

Let's take a look at the following example:

```
...
<p><em>Bear</em>! Who the hell is this <em>Bear</em>!</p>
<p><strong>Caution!</strong> <em>Bear</em> could be standing behind you!</p>
<p><strong>Delivery date in <strong><em>summer 2022</em></strong></strong></p>
...
```

In this example, it's semantically clear from the emphasis on `bear` in the first paragraph in Figure 4.38 that it isn't the animal that is referred to here, but a person with the surname "Bear". In the second paragraph, the word `Caution!` was marked with a special importance. In the last paragraph, an `em` element was nested within a `strong` element to emphasize `summer 2022` more strongly in terms of content in addition to its particular importance.



Bear! Who the hell is this *Bear*!

Caution! *Bear* could be standing behind you!

Delivery date in *summer 2022*

Figure 4.38 Different Ways to Emphasize or Highlight a Text Using <em> and <strong>

Here, you could also still nest the same HTML elements to increase the emphasis or importance of `em` or `strong` elements, at least semantically.

Because web browsers usually render `<em>` with italic and `<strong>` with bold font, you shouldn't make the mistake of replacing these elements with `<i>` and `<b>`, respectively, because these elements (i.e., `<em>` and `<i>` or `<strong>` and `<b>`) are rendered quite similarly in the web browser. As mentioned at the beginning, HTML isn't used to format the text, but these HTML elements are about giving the text a meaningful significance.

The `i` element is recommended to mark special technical terms, a thought, scientific names, or foreign language words. The `b` element, on the other hand, should be used for meaningful names or keywords to draw attention to something.

Using <i> and <b>

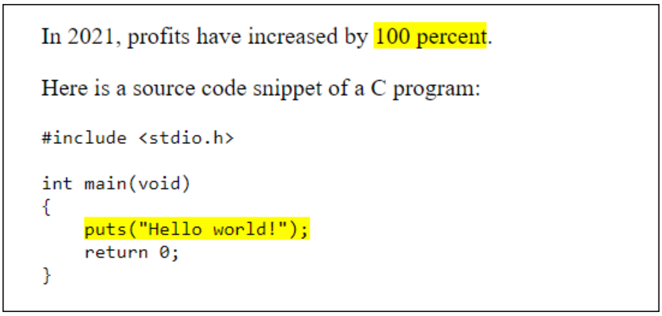
Standard HTML recommends using the `b` or `i` elements only if no other HTML tag fits to mark up the text or passage. The days when these elements were used purely for formatting are over because CSS is used for that purpose.

4.4.9 Highlighting Text Using <mark>

You should use the `mark` element to mark up words or passages in a continuous text. The easiest way to compare such a markup is with a highlighter. In practice, this HTML element should be suitable for visually highlighting the search term found during a search. This works only if the content is generated dynamically. The element is also very suitable for highlighting code fragments of a source code.

The following code snippet demonstrates the `mark` element whose execution is shown in Figure 4.39:

```
...
<p>In 2021, profits have increased by
   <mark>100 percent</mark>.
</p>
<p>Here is a source code snippet of a C program:</p>
<pre><code>#include &lt;stdio.h&gt;

int main(void)
{
    <mark>puts("Hello world!");</mark>
    return 0;
}</code></pre>
...
```



In 2021, profits have increased by 100 percent.

Here is a source code snippet of a C program:

```
#include <stdio.h>

int main(void)
{
    puts("Hello world!");
    return 0;
}
```

Figure 4.39 Web Browsers That Recognize the New Element Usually Mark the Text Placed between <mark> and </mark> with Yellow Background Color

Even though the `mark` element predominantly applies yellow background color to the text, you shouldn't use it to apply a background color to a text. You should rather use `<mark>` only if it makes sense in terms of content and no other semantic HTML element is suitable. If you want to format the text background, you should use the `span` element (Section 4.4.17) with CSS instead.

4.4.10 Placing Text between Quotes Using <q>

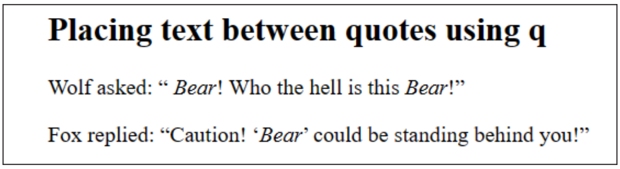
While you can use the `blockquote` element to quote an entire paragraph text, the `q` element allows you to quote something in the middle of the text or mark it as spoken text. Text or passages you insert between `<q>` and `</q>` should be placed between quotes by the web browser. You shouldn't use the `q` element if you simply want to enclose a word or passage in quotation marks. That wouldn't be the semantic sense of the `q` element.

If you use a quote or spoken text from another source, you can use the `cite` attribute with a URL to the source. Because the implementation of the `cite` attribute is still poor in web browsers, you may want to consider using a hypertext link.

You can also nest the q element. Such nested q elements usually get another matching quotation mark. In this country, for example, the outer quotation marks are double and the inner ones are single. This is demonstrated in the following example, the execution of which you can see in Figure 4.40:

```
...
<p>Wolf asked: <q cite="http://tom-bear.com/">
   <em>Bear</em>! Who the hell is this <em>Bear</em>!</q>
</p>
<p>Fox replied: <q>Caution! <q><em>Bear</em></q> could be
   standing behind you!</q>
</p>
...
```

As you can see in Figure 4.40, this example quoted spoken text in the first paragraph. The second paragraph demonstrates the nesting of q elements. The inner q element was placed between single quotes and the outer one between double quotes.



Placing text between quotes using q

Wolf asked: “ *Bear*! Who the hell is this *Bear*!”

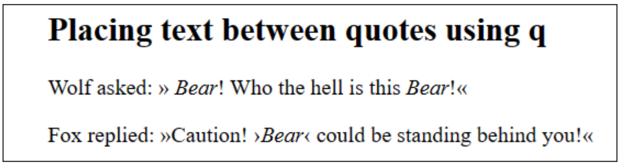
Fox replied: “Caution! ‘*Bear*’ could be standing behind you!”

Figure 4.40 Placing Text between Quotes Using the <q> Element

For Advanced Users: Setting Custom Quotation Marks

The problem with the q element is that the quotes set by the web browser may not always be the ones you want. In this case, an intervention with CSS is a good idea. Thus, the following CSS line could be used to change the first and second nested characters of the q element:

```
q {quotes: '»' '«' '›' '‹';}
```



Placing text between quotes using q

Wolf asked: » *Bear*! Who the hell is this *Bear*!«

Fox replied: »Caution! ›*Bear*‹ could be standing behind you!«

Figure 4.41 The Quotes of the <q> Element Have Been Changed with CSS

4.4.11 Underlining or Crossing Out Text Using <u> and <s>

Both the u element (u = *underline*) and the s element (s = *strikethrough*) were marked as deprecated with HTML 4.01 and were supposed to be removed from the standard. With the new HTML standard, they have acquired a new semantic meaning and are thus again an official part of HTML.

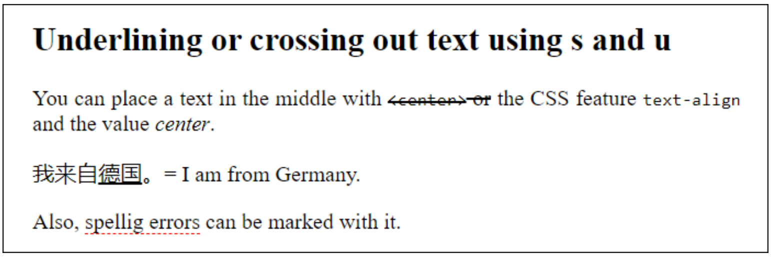
You should use the `s` element to mark content as obsolete or no longer correct. The web browsers display the text between `<s>` and `</s>` as strikethrough. If you want to display a document edit where you want to mark a word or passage from the not yet finished document as deleted, you should use the `del` element instead.

According to its new meaning, the `u` element is to be used for underlining proper names, as is common, for example, in Chinese writing (see *http://en.wikipedia.org/wiki/Proper_name_mark*). Most readers are unlikely to use Chinese proper names, so another recommended example of the `u` element is to knowingly indicate misspelled words or passages containing errors. Web browsers usually display the `u` element with an underscore. In addition to the `u` element, there's the `ins` element, which is also rendered as underlined but is intended to indicate newly inserted content (Section 4.4.12).

Here's a short example, the execution of which you can see in Figure 4.42:

```
...
<p>You can place a text in the middle with
   <s><code>&lt;center&gt;</code> or</s> the
   CSS feature <code>text-align</code> and the value
   <var>center</var>.
</p>
<p>我来自
<u>德国
</u>。 = I am from Germany.</p>
<p>Also, <u  class="spell-checker">spellig errors</u>
   can be marked with it.
</p>
...>
```

As you can see in Figure 4.42, in the first paragraph text, the content `<center> or` was crossed out. In the second paragraph text, a Chinese proper name (Germany = 德国) is underlined. Finally, in the final paragraph, a spelling error was underlined with a red dashed line. The color changes and the style of the underline were adjusted with CSS in the example.



Underlining or crossing out text using s and u

You can place a text in the middle with ~~<center> or~~ the CSS feature `text-align` and the value *center*.

我来自德国。 = I am from Germany.

Also, spellig errors can be marked with it.

Figure 4.42 Underlining or Crossing Out Text Using <u> and <s>

In any case, it's important that you use both elements for their intended content and not for text decoration. If you want to underline or strike through text for purely decorative reasons, you should use CSS to do so.

4.4.12 Marking Changes of Text Using <ins> and <del>

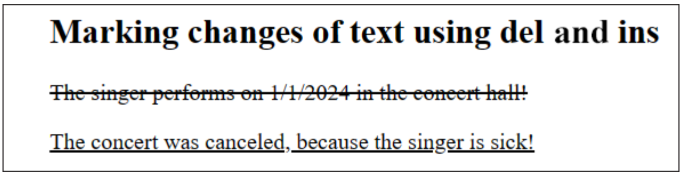
The `ins` element is rendered by web browsers similar to the `u` element, while the `del` element is rendered similar to the `s` element. Nevertheless, the semantic meaning of the two elements is different and therefore not interchangeable.

The `del` element (`del` = *delete*) allows you to mark a content-related (active) editing of a text as deleted in the revised sense. It's used to inform readers that this part has been revised or further developed. Instead of removing the text completely, you want to make sure that the previous versions of the text remain visible. The web browser usually crosses out this text.

The `ins` element (`ins` = *insert*) is the counterpart of the `del` element and should be used when something new gets inserted into the document. Here again, you mark a further development of the previous version of the text. The web browsers usually display this text with an underscore.

You can see the execution of the following example with the elements `del` and `ins` in Figure 4.43.

```
...
<del>
  <p>The singer performs on 1/1/2024 in the concert hall!</p>
</del>
<ins>
  <p>The concert was canceled,
     because the singer is sick!</p>
</ins>
...
```



Marking changes of text using del and ins

~~The singer performs on 1/1/2024 in the concert hall!~~

The concert was canceled, because the singer is sick!

Figure 4.43 The <del> Element Used to Delete a Paragraph Text and Insert a New Paragraph with a New Message between <ins> and </ins>

Again, you should use the `del` and `ins` elements only if they fit the semantics. If you want to cross out or underline the text in a purely decorative way, CSS should be the first choice.

Text Underline Can Be Confusing!

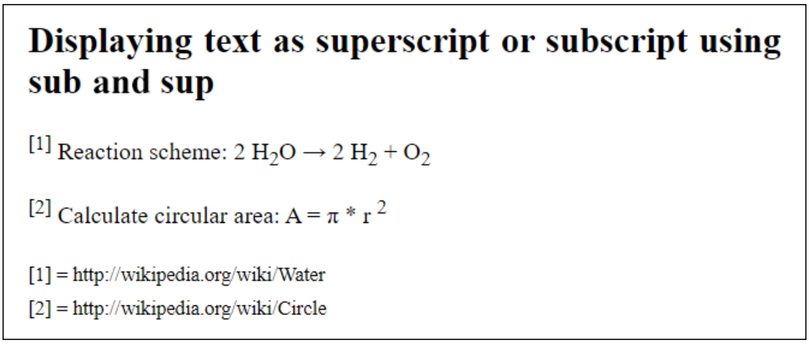
The frequent use of underscores with `<ins>` or `<u>` may confuse the user because hypertext links with `<a>` are usually also represented with an underscore. It's therefore also recommended to change the formatting with CSS, so that the individual elements are displayed in a clearly distinguishable manner.

4.4.13 Displaying Text as Superscript or Subscript Using <sup> and <sub>

These two markups are mainly used for simple mathematical and chemical formulas to lower text with the `sub` element (`sub` = *subscript*) and to raise it with the `sup` element (`sup` = *superscript*).

Here's a code snippet as an example, the execution of which you can see in Figure 4.44:

```
...
<p><sup>[1]</sup> Reaction scheme: 2 H<sub>2</sub>O
   &RightArrow; 2 H<sub>2</sub> + O<sub>2</sub></p>
<p><sup>[2]</sup> Calculate circular area: A = &pi; * r
   <sup>2</sup></p>
...
```



Displaying text as superscript or subscript using sub and sup

[1] Reaction scheme: 2 $H_2O \rightarrow 2\ H_2 + O_2$

[2] Calculate circular area: $A = \pi * r^2$

[1] = http://wikipedia.org/wiki/Water
[2] = http://wikipedia.org/wiki/Circle

Figure 4.44 The <sub> and <sup> Elements Were Used Several Times for Superscript and Subscript Numbers and Footnotes, Respectively

4.4.14 Marking Dates and Times Using <time>

The `time` element was introduced to mark up dates and times. When displayed in a web browser, text placed between `<time>` and `</time>` is usually not visually noticeable at all. The goal and purpose of the `time` element is rather that date and time are uniquely coded for machines and can also be displayed in a readable way for humans. You can specify the machine-readable form in the HTML attribute `datetime`, while the human-readable form is usually written between `<time>` and `</time>`. Here's a brief example:

```
...
<p>We met on my <time datetime="2023-11-12">
   44th birthday</time> at <em>Bear's place</em>.
   <time datetime="20:00">at 8 pm.</time>
</p>
...
```

The specification `44th birthday` can be any other text such as *Wednesday* or *November 12* as long as the value of `datetime` is a precise date of the Gregorian calendar. The same applies to the time `at 8 pm`.

Valid Machine-Readable Date and Time Information

A date readable by machines is specified as YYYY-MM-DD. YYYY is the year (e.g., 2023), MM is the month (e.g., 11 for November), and DD is the day in the month (e.g., 12). If you also want to note the time, you must place a capital `T` between the date and the time and then enter the time behind it in the form HH:MM where HH stands for hours and MM for minutes. In a newer version of `time`, this `T` can be omitted and a space can be used instead. Optionally, you can specify the time zone offset from UTC (*Coordinated Universal Time*). UTC is a designation for Universal Time. Here, you must specify a + followed by HH:MM, for example:

```
2023-10-10T21:00+01:00
```

For this example, you enter October 10, 2023, as the date. The time is exactly 9 pm, and the +01:00 at the end means UTC + 1 hour. Many other different forms of presentation are possible in this context. For more information, you should visit *www.w3.org/TR/2011/WD-html5-20110525/text-level-semantics.html#the-time-element*. For further and future developments of the `time` element, you may find the following website useful: *http://wiki.whatwg.org/wiki/Time_element*.

Alternatively, you can specify the date, time, and time zone (if needed) in `datetime` at once:

```
...
<p>We met on my
   <time datetime="2023-11-12T20:00+00:00">44th birthday
   </time> at <em>Bear's place</em> at 8pm.
</p>
...
```

So, if you use a valid `datetime` attribute with the `time` element, you can write whatever you want between `<time>` and `</time>`. *Without* specifying the `datetime` attribute, you *must* specify a valid date format and/or a valid time format—that is, the machine-readable version—between `<time>` and `</time>`, such as the following:

```
...
<p>We met on <time>2023-11-12</time> at <em>Bear's place</em>
   at <time>20:00</time>
</p>
...
```

The `time` element has been improved and made much more flexible over time after its first release. For example, the following specifications are also possible:

```
...
<p>On every <time datetime="11-12">birthday</time>
   I got flowers.
</p>
...
```

This refers to November 12. Another option would be to specify the following for `datetime` if you don't remember the exact day of the date:

```
<p>The concert in the photo was recorded sometime in
   <time datetime="2023-08">August</time>
   this year.
</p>
```

Here, a date in August 2023 is meant. It's possible to use only the year (e.g., `datetime="2023"`).

Another improvement is that you can use a time duration. Here's an example of how such a duration is represented:

```
<p>the rock festival lasted <time datetime="P3D">3
   days</time>.
</p>
```

The letter `P` stands for *period*, the `D` for *day*, and the `3` for three days. You can still specify a time period from a combination with `H` for *hours*, `M` for *minutes*, and `S` for *seconds* (e.g., `datetime="P1D5H10M"` = 1 day, 5 hours, and 10 minutes).

Here's What Doesn't Work (Yet)!

It isn't yet possible to specify a time before Christ (BC), neither can you specify a time period based on two date ranges. For this purpose, you still need to use two `time` elements.

What Should I Use <time> for in Practice?

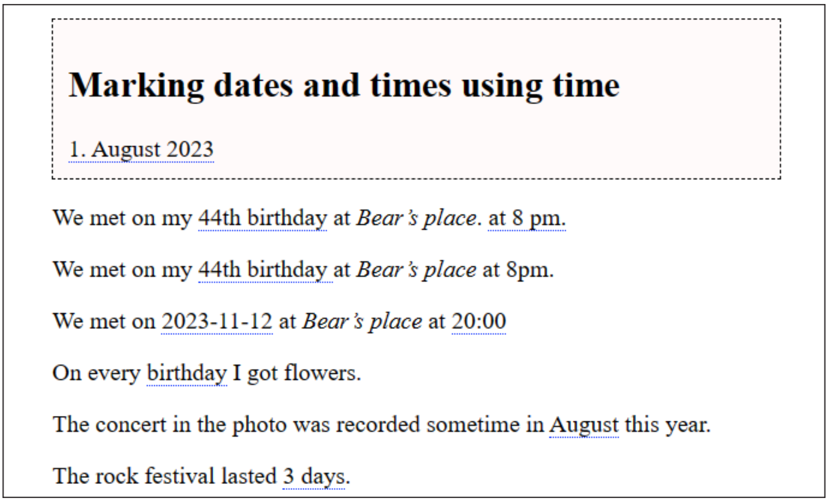
If you use the `time` element, it will be easier for other programs to index this data. For example, it's easier for a script in blog articles to extract the date using the `time` element, rather than using any other techniques to look for and read this data. By having the date and/or time information in a machine-readable form, there's the advantage for search engines to make use of it when searching for items of a certain period or date.

If you then also use the `datetime` attribute, you can provide readers with a reader-friendly alternate display between `<time>` and `</time>`. Basically, you should put a (readable) date and/or time between `<time>` and `</time>` because the web browser won't display an automatic value in between if you don't put anything there.

Another possibility is that future web browsers provide the option to enter a date into the calendar at the request of the visitor. In addition, the web browser could convert the time used into the visitor's time zone if the appropriate value was specified with `datetime`. And he could convert the season according to the Buddhist time calculation, which is valid, for example, in Thailand or Laos.

In Figure 4.45, you can see again all `time` elements described and used here. I underlined the places where `<time>` was used with a dotted line using CSS for better visibility. Here, the date on which the article was written was additionally indicated behind the article title. To use the `time` element in a semantically correct way, the specification of an exact time must be used and observed. For example, you should avoid the following incorrect usage:

These are the results from `<time>last week</time>`.



Marking dates and times using time

1. August 2023

We met on my 44th birthday at *Bear's place*. at 8 pm.

We met on my 44th birthday at *Bear's place* at 8pm.

We met on 2023-11-12 at *Bear's place* at 20:00

On every birthday I got flowers.

The concert in the photo was recorded sometime in August this year.

The rock festival lasted 3 days.

Figure 4.45 To Clarify What Is between <time> and </time>, a Dotted Underline Has Been Added

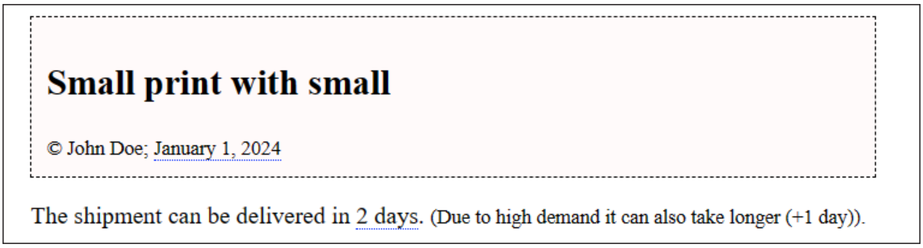
4.4.15 Marking the Small Print Using <small>

You should use the `small` element for words or text passages in which you want to display some small print. This can be copyright information, license information, legal notes, and so on.

Here's a short code snippet with the `small` element; its execution is shown in Figure 4.46.

```
...
<article>
  <header>
    <h2>Small print with small</h2>
    <small>&copy; John Doe;
    <time datetime="2024-01-01">January 1, 2024</time></small>
  </header>
  <p>The shipment can be delivered in <time datetime="P2D">2 days</time>.
     <small>(Due to high demand
     it can also take longer (+1 day)).</small>
  </p>
</article>
...
```

Again, you should use this element only if it semantically fits the content and not to make text look smaller. For visual adjustments, you should use CSS.



Small print with small

© John Doe; January 1, 2024

The shipment can be delivered in 2 days. (Due to high demand it can also take longer (+1 day)).

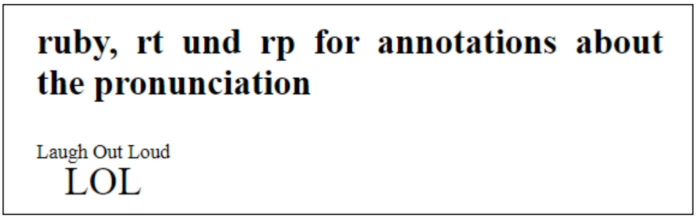
Figure 4.46 A Copyright Was Placed in the Head of an Article as well as Small Printed Information between <small> and </small>

4.4.16 Using <ruby>, <rp>, and <rt> for Annotations about Pronunciation

The Ruby annotation is probably of little interest to most readers. This is an annotation system in which the text appears with the annotation in one line, as used in Japanese and Chinese texts. If you want to know more about this notation, see *https://en.wikipedia.org/wiki/Ruby_character*.

Here, we'll only briefly describe the use of the Ruby annotation with the existing HTML elements. For this purpose, here's a simple example in which Asian characters have been omitted:

```
...
<p>
  <ruby>
    LOL<rp> (</rp><rt>Laugh Out Loud</rt><rp>)</rp>
  </ruby>
</p>
...
```

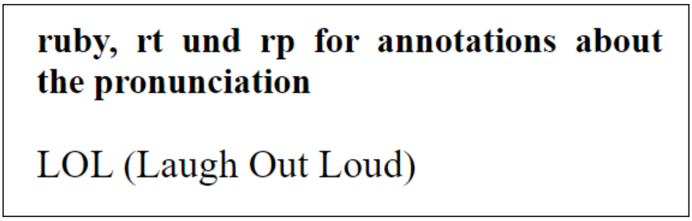


ruby, rt und rp for annotations about the pronunciation

Laugh Out Loud
LOL

Figure 4.47 The Text between a Ruby Annotation Is Displayed as Text with Annotation in One Line

The example features `LOL` with annotations (which is nonsense here, of course). All characters, including annotations, are written between `<ruby>` and `</ruby>`. Then the character is noted as element content (here: LOL). The parentheses of the annotations are created with the (optional) `rp` element (`rp` = *Ruby parenthesis*). The text is then marked with the `rt` element (`rt` = *Ruby text*). Thus, the annotation is written between `<rt>` and `</rt>`.



ruby, rt und rp for annotations about the pronunciation

LOL (Laugh Out Loud)

Figure 4.48 The Optional <rp> Element Is Used to Put Parentheses around the Ruby Text (with the <rt> Element) for Web Browsers That Don't Understand <ruby>

Web Browser Support for <ruby>

Web browser support in current web browsers is now quite good. In some web browsers, however, an add-on may need to be installed.

4.4.17 Grouping Ranges of Individual Text Passages Using <span>

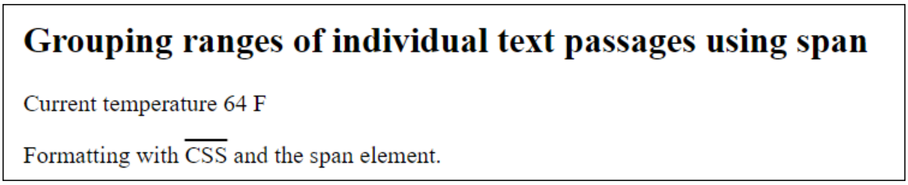
While you can use the `div` element (Section 4.2.7) to group entire groups into one block, you can use the `span` element to mark up individual passages of text inside the body text with CSS. Visually, text placed between `<span>` and `</span>` doesn't change at all. In addition, the `span` element in conjunction with a JavaScript is quite useful when searching for global attributes used within it to directly update the element content. Here's a rather theoretical example:

```
<p>Current temperature <span id="temp">64</span> F</p>
```

Here, you could use a JavaScript to read the global `id` with the value `temp` and further process or update the element content. Let me also show you this CSS example:

```
<p>Formatting with <span style="text-decoration:overline;">
   CSS</span> and the span element.</p>
```

Here, the text is preceded by a stroke that has been implemented via style statements directly in the HTML tag `span` with the HTML attribute `style`.



Grouping ranges of individual text passages using span

Current temperature 64 F

Formatting with CSS and the span element.

Figure 4.49 The <span> Element Has No Default Formatting; Besides Designing with CSS, It Can Also Be Used to Identify Unique Elements.

However, as with the `div` element, you should only use this element if there's no semantically more suitable HTML element available.

Example of All Text Markup Elements

Once again, note that you can find an example with all the text markup elements demonstrated here on the web page for the book (*www.rheinwerk-computing.com/5695/*) and at *https://html-examples.pronix.de/examples/chapter004/4_4/index.html*.

4.5 Related Topic: Character Encoding

What follows here isn't a treatise on character encoding in general, but rather in the context of HTML documents—especially on how you can avoid special characters, such as German umlauts, from getting displayed as cryptic characters. If you consider the following two points, there shouldn't be any problem:

- In the HTML document, you need to specify the character encoding of the document in the head data between `<head>` and `</head>`, as we've been doing in the book so far with `<meta charset="UTF-8">`. Unless you have a specific reason, you should always use the `UTF-8` value for the `charset` attribute.

- However, it doesn't suffice to specify the character encoding in the HTML document as it must also be saved in this encoding using the editor of your choice. Consequently, if you've specified UTF-8 as the character encoding in the HTML document, you must also save the document with the UTF-8 encoding. With most editors, you no longer have to bother about this. Nevertheless, it should be briefly mentioned here.

4.5.1 From Bytes to Character Encoding

The smallest unit, the bit, will be skipped here because you don't need to go so deep into detail at this point. The byte unit is quite sufficient for this purpose. When the computer reads a file or data into the main memory, it's basically just bytes that have a certain value. The value of a single byte results from the states of the individual bits. Let's use a byte with the value 68 (incidentally, with the bit value 1.000.100) as an example.

To create a human-readable character from this byte with the value 68, a convention is needed that describes which byte value corresponds to which representable character. For this purpose, a translation table (also referred to as encoding table) is used for encoding bytes.

4.5.2 From ASCII to ISO-8859

You know that an encoding table is responsible for turning a byte into a readable character. The first type of such a character set was introduced with the ASCII encoding and the EBCDIC encoding, with which 128 different states can be represented on 7 bits. ASCII encoding has become established in common practice. In the ASCII encoding table, the first 32 characters are pure control characters, and the actual characters are stored in the character set between 32 and 127. A look at the ASCII encoding table shows that the value 68 corresponds to the capital letter D.

The 8th bit was initially used only for error-correction purposes (parity bit) for communication lines or other control tasks. Because there was no space left in the ASCII character set between the values 32 and 127 for language-specific characters (e.g., umlauts), the 8th bit was used to extend the character set. At this point, the Babylonian character confusion also arose because different developers wrote their own "8th-bit-codes." IBM PCs and English MS-DOS systems used codepage 435, for example. In Germany, codepage 850 was used for Western European characters.

Newer standards such as ISO-8859 also use 8 bits. Here, several character set tables were developed at once. For example, ISO-8859-1 (or Latin-1) represents the Western European languages. The first 127 characters were taken over from the ASCII encoding. In the values between 128 and 255, many special characters and important characters from different European languages were implemented (with the German umlauts, the Spanish tilde character, or the French accent characters).

So, theoretically, you can use the ISO-8859-1 character set for the HTML document:

```
<meta charset="iso-8859-1">
```

While in theory, you can use any character set for `charset`, you should keep in mind that not every web browser understands all character sets. If you use a more widely used character set, you have a better chance that a web browser in distant countries will be able to do something with it.

Microsoft had also added its own variation to the ISO-8859-1 encoding with codepage 1252. After all, code page 1252 already contained the euro sign. ISO-8859-1, on the other hand, doesn't recognize the euro sign because at the time this table was created, the euro didn't even exist. The euro sign was added by the ISO only later with ISO-8859-15. Now the situation here is that ISO-8859-1 doesn't recognize the euro sign, while ISO-8859-15 and codepage 1252 do know it, but the value in the encoding table is again different. Fortunately, today you don't need to deal with the different character sets of a language. The description of the ISO 8859 standards here serves only as background information on the subject.

The current HTML specification uses the Unicode UTF-8 character set with `charset="UTF-8"`.

4.5.3 Beyond the Byte Boundary with Unicode

The preceding provided a good impression of the confusion regarding the different character encodings. Note, however, that this was only about the Western European character set, and I haven't really gone into detail yet. The fact is that character encoding can be relatively complex if you pack everything into a byte and then want to use different characters from different cultures. To bring all characters under one hood, the Unicode system was introduced.

The Unicode character set can be used to represent all human-made characters. In purely theoretical terms, more than four billion characters could be used with 32 bits per character—in practice, Unicode is limited to about one million code points. UTF-8 is the 8-bit encoding of Unicode, which is also backward compatible with ASCII encoding. A character can contain between one and four 8-bit words. UTF-8 is now a uniform standard. For example, many operating systems use UTF-8 by default, and UTF-8 is also being used increasingly in web development with HTML to represent language-specific characters, where it more and more replaces the use of HTML entities (Section 4.6).

More Information Online

I could write much more on this topic, especially Unicode, but this would go beyond the scope of this book. For more information, visit *http://r12a.github.io/scripts/tutorial/* and *https://home.unicode.org/*. You can also find the characters of the Unicode encoding at *www.unicode.org/charts/*.

4.6 Character Entities in HTML

While the importance of character entities in HTML has diminished considerably with the gradual spread of Unicode (especially UTF-8), I should still touch on them briefly here because there are always reasons to use them. In Section 4.5, you learned about different character sets, and, by now, you know how to specify the character set used as a `<meta>` tag in the HTML document head with `charset`. For example, if you've specified ISO-8859-1 or ISO-8859-15 as the character set and want to use the word shalom (=שלום) in Hebrew characters, you're likely to be unsuccessful:

```
<meta charset="iso-8859-1">
  ...
<p>Shalom: שלום</p>
```

The output is likely to be a cluster of cryptic characters instead of שלום. The simplest solution would be to change the character set to UTF-8 via

```
<meta charset="UTF-8">
```

There might also be a different problem with this example: How do you type the word שלום in your editor? If you don't happen to have a Hebrew keyboard in front of you or a virtual keyboard with Hebrew characters, the simple and quick solution might be to use the character entities of HTML. This is how you write the word "Shalom" in Hebrew using character entities:

```
<p>Shalom: <bdo>&#1501;&#1503;&#1500;&#1513;</bdo></p>
```

4.6.1 Structure of a Character Entity in HTML

As you've seen before from the four Hebrew characters, an HTML entity starts with the `&` character and ends with the semicolon. Now you have two options to arrange the sign:

- **Numeric entities**
  You specify the form with `&#nnn;`. Here, **nnn** stands for the encoding of the character. This form is used when it isn't possible to enter the character via the keyboard. The

notation can also be in the form of `&#xhhh;`, where **xhhh** is the hexadecimal value for the character. The notation without **x** is the decimal notation.

- **Named entities**
  This is an easier-to-remember name that has been agreed on for the character. You may have already seen examples with `&lt;` (`lt` = *less than*) or `&gt;` (`gt` = *greater than*) where people prefer to use these named entities. Alternatively, you can use the numeric entity instead of the named entities. For example, with `&lt;`, `&#60;`, and `&#x003C;`, you would use the < sign (less-than sign) three times.

Masking HTML-Specific Characters

Especially if you use special characters in your body text that are part of the HTML syntax, you should mask these characters by using the appropriate entity. For example, the following line is likely to cause display problems in a web browser:

```
<p>Mexico City<Tokyo and Mumbai>London</p>
```

The web browsers would only output `Mexico City-London` here, because the area between < and > is considered an HTML element (even if it's wrong). Although you could solve this problem with a blank line in between, you should use the appropriate entity for this, to be on the safe side. This is where the named entity comes in handy:

```
<p>Mexico City&lt;Tokyo and Mumbai&gt;London</p>
```

The ampersand character & belongs to it as well and should be used via the string `&amp;` in the continuous text.

In addition, if you want to use the double quote within HTML attributes, you should replace " with `&quot;`, such as the following:

```
<img src="#" alt="Cover of book &quot;CSS from A to Z&quot;">
```

In the `alt` attribute, `&quot;` was used as a masking character for ". If you used " instead of the named entity `&quot;` here, the area in between would probably be "swallowed" by the web browser.

More Unicode Numbers

You can find even more Unicode numbers for a desired character at *www.unicode.org/charts*.

4.7 Summary

In this chapter, you learned a lot about the semantic use of various HTML elements. Roughly summarized, you've learned the following in this chapter:

- How to use the `<section>`, `<article>`, `<aside>`, and `<nav>` elements to divide an HTML document into meaningful parts (or sections)
- How to use headings with the elements `<h1>` to `<h6>` and the new section elements `<section>`, `<article>`, `<aside>`, and `<nav>` to affect the content structure of headings
- How to use a header with `<header>` and a footer with `<footer>` in an HTML document
- How to use the `<main>` element to set the main content of a web page
- Which HTML elements are available to divide or group plain text content into paragraphs
- What semantic HTML is and how you can structure a semantic web page
- How to logically mark up text, individual letters, words, or parts of sentences to give them semantic meaning
- That the HTML elements for text markup aren't used for formatting web pages, but that this is done via CSS

Chapter 5

Tables and Hyperlinks

This chapter introduces you to more HTML elements. More specifically, you'll learn how to add and use tables and hyperlinks.

This chapter describe other essential HTML elements that haven't been dealt with up to now. In particular, you'll learn more about the following topics:

- **Tables**
  You'll learn how to use tables to represent information or data in a grid.
- **Hyperlinks**
  Every internet user is familiar with hyperlinks that allow them to move from one website to another. You'll learn how to link an HTML document to other HTML documents.

5.1 Structuring Data in a Table

Tables are useful when you want to display data such as stock quotes, financial information, travel schedules, train schedules, bus schedules, travel reports, or sports scores in a grid of rows and columns. HTML provides some viable options to structure such a table, as listed in Table 5.1.

HTML Element	Meaning
`<table>`	Table
`<tr>`	Table row
`<td>`	Table cell
`<th>`	Table header cell for heading
`<thead>`	Table header area
`<tbody>`	Table body
`<tfoot>`	Table foot section
`<colgroup>`	Group of table columns

Table 5.1 Brief Overview of the Table Elements Covered Here

HTML Element	Meaning
`<col>`	Table column
`<caption>`	Table header/legend

Table 5.1 Brief Overview of the Table Elements Covered Here (Cont.)

Formatting with CSS

HTML is used only for semantic and structural logical markup, and this is also true for tables in HTML. Tables in HTML don't provide any formatting options. All attributes for formatting from old HTML, except for `border`, are no longer supported by the current standard HTML version. For this reason, the same applies here: tables should be formatted using CSS.

5.1.1 A Simple Table Structure Using <table>, <tr>, <td>, and <th>

Every table in HTML is created between the elements `<table>` and `</table>`. The contents of the table are written row by row. The beginning of a row must contain an opening `<tr>`, while the row must end with a closing `</tr>` (`tr` = *table row*). Within a table row between `<tr>` and `</tr>`, you write the individual cells (or also columns) with `<td>` and `</td>` (`td` = *table data*).

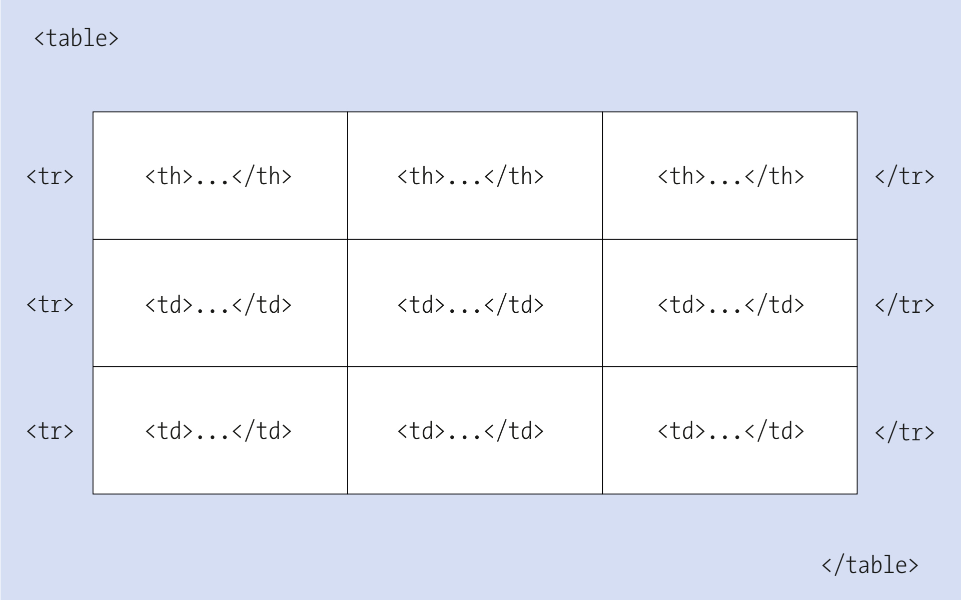


Figure 5.1 A Basic Table Structure in HTML

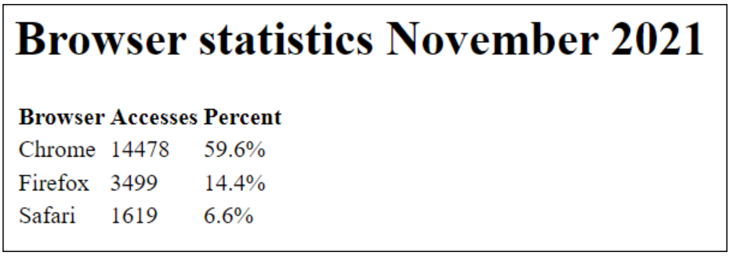
If you want to use cells or columns as headers of a table, you can place the data between `<th>` and `</th>` (`th` = *table heading*). You can use the `th` element in the same way as the `td` element, except that web browsers usually highlight this element with a bold font centered in the column. If it makes sense, you should use table headings, as this is helpful for visitors with screen readers and, if applicable, for search engines, which can index your website better with table headings.

For this purpose, we want to create a simple example of a table in which web browser statistics data from a website is summarized in a grid and displayed in a clear overview:

```
...
  <table>
    <tr>
      <th>Browser</th>
      <th>Accesses</th>
      <th>Percent</th>
    </tr>
    <tr>
      <td>Chrome</td>
      <td>14478</td>
      <td>59.6%</td>
    </tr>
    <tr>
      <td>Firefox</td>
      <td>3499</td>
      <td>14.4%</td>
    </tr>
    <tr>
      <td>Safari</td>
      <td>1619</td>
      <td>6.6%</td>
    </tr>
  </table>
...
```

Listing 5.1 /examples/chapter005/5_1_1/index.html

As you can see in Figure 5.2, web browsers display the table without any formatting. The height and width of a table are usually adjusted to its contents.



Browser statistics November 2021

Browser	Accesses	Percent
Chrome	14478	59.6%
Firefox	3499	14.4%
Safari	1619	6.6%

Figure 5.2 The Structured Representation of a Basic Table in HTML

What Is Allowed in a Table Cell?

In a cell between `<td>` and `</td>`, you can use other HTML elements in addition to text. Theoretically, you could insert another complete table into it. If you want to use an empty cell without content, you must still specify an empty `<td></td>` or `<th></th>`; otherwise, the table won't be displayed correctly. In old web browsers, you can also write a forced space with the HTML entity `&nbsp;` in the cell to be on the safe side because there could be problems with empty cells.

5.1.2 Combining Columns or Rows Using "colspan" or "rowspan"

If you want to combine (or span) table entries across multiple cells, you can do this using the HTML attributes `colspan` and `rowspan`. Based on the numerical value you pass to these attributes, the number of cells you want to merge is specified. As you might guess from the attribute names, `colspan` is used to group columns together, and `rowspan` is used to group rows together.

Here's a simple example in which the daily schedule of a photography seminar was summarized in a table:

```
...
  <table>
    <tr>
      <th></th>
      <th scope="col">Morning</th>
      <th scope="col">Noon</th>
      <th scope="col">Afternoon</th>
    </tr>
    <tr>
      <th scope="row">Monday</th>
      <td colspan="2">Photo shooting (outdoor)</td>
      <td>Image editing workshop</td>
    </tr>
    <tr>
```

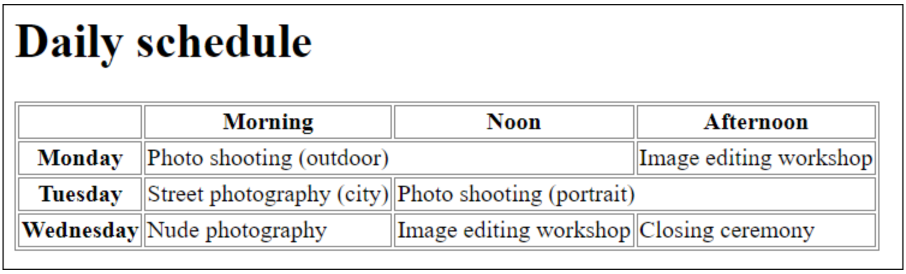
```
      <th scope="row">Tuesday</th>
      <td>Street photography (city)</td>
      <td colspan="2">Photo shooting (portrait)</td>
    </tr>
    <tr>
      <th scope="row">Wednesday</th>
      <td>Nude photography</td>
      <td>Image editing workshop</td>
      <td>Closing ceremony</td>
    </tr>
  </table>
...
```

Listing 5.2 /examples/chapter005/5_1_2/index.html

As you can see in Figure 5.3, CSS was used to frame the table so that the result of `colspan` is more visible.

Here, you can see how, on Monday, the `Photo shooting (outdoor)` cell spans both the `Morning` and `Noon` columns thanks to `colspan="2"`. The same is true for Tuesday and the column `Photo shooting (portrait)`, where `Noon` to `Afternoon` has been combined.

When using `colspan`, you must keep in mind that you need to reduce the number of cells if, for example, you combine a `colspan` across two cells. In the `Monday` example, you thus only need to write two `td` elements instead of three because the first `td` element already spans two columns.



Daily schedule

	Morning	Noon	Afternoon
Monday	Photo shooting (outdoor)		Image editing workshop
Tuesday	Street photography (city)	Photo shooting (portrait)	
Wednesday	Nude photography	Image editing workshop	Closing ceremony

Figure 5.3 Merging Columns Using the "colspan" Attribute

By the way, there's nothing against merging more than two columns. Here, you must pay attention to the number of columns that are actually present. As an example, on `Tuesday`, you could merge the `Photo shooting (portrait)` across three columns:

```
...
    <tr>
      <th scope="row">Tuesday</th>
      <td colspan="3">Photo shooting (portrait)</td>
    </tr>
    <tr>
...
```

However, the `Street photography (city)` cell would then have to be removed as well.

The "Scope" Attribute of <th>

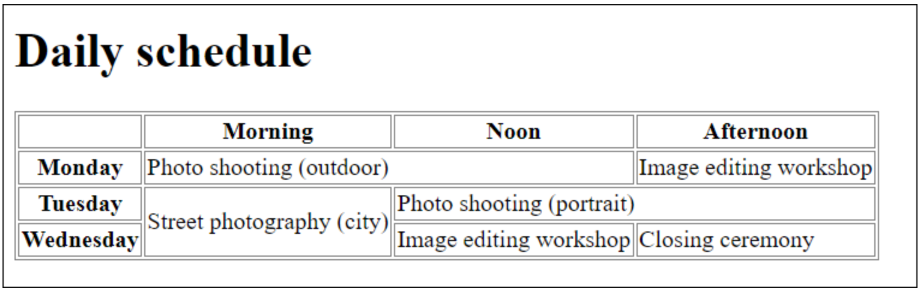
In the example, the `scope` attribute was used with the `th` element. This allows you to specify whether the table heading should apply to a column (`scope="col"`) or a row (`scope="row"`).

Everything just described also applies if you want to combine table entries across multiple rows using `rowspan`. For this purpose, here's the example again in which the daily schedule for the photo seminar has been changed a bit because now `Street photography (city)` takes place in the morning on `Tuesday` and `Wednesday`:

```
...
  <table>
    <tr>
      <th></th>
      <th scope="col">Morning</th>
      <th scope="col">Noon</th>
      <th scope="col">Afternoon</th>
    </tr>
...
      <th scope="row">Tuesday</th>
      <td rowspan="2">Street photography (city)</td>
      <td colspan="2">Photo shooting (portrait)</td>
    </tr>
    <tr>
      <th scope="row">Wednesday</th>
      <td>Image editing workshop</td>
      <td>Closing ceremony</td>
    </tr>
  </table>
...
```

Listing 5.3 /examples/chapter005/5_1_2/index2.html

In the last `tr` element, you need to remove the `td` element with `Nude photography` because you've extended the `Street photography (city)` entry above it downward using the `rowspan` attribute, which causes that entry to take up space in the cell below it, as you can see in Figure 5.4.



Daily schedule

	Morning	Noon	Afternoon
Monday	Photo shooting (outdoor)		Image editing workshop
Tuesday	Street photography (city)	Photo shooting (portrait)	
Wednesday		Image editing workshop	Closing ceremony

Figure 5.4 Merging Rows Using the “rowspan” Attribute

5.1.3 HTML Attributes for the Table Elements

For the `table` element, HTML supports only the `border` attribute to indicate a border; the value can be `"1"` or `""`. CSS is recommended as the better option here. For example, to copy `border="1"`, you can simply add the following CSS construct to the HTML document head:

```
...
<style>
  table, td, th { border: 1px solid gray }
<style>
...
```

There are no attributes for the table row with `<tr>`. You’ve already learned about the attributes of `<td>` and `<th>` with `colspan`, `rowspan`, and `scope`.

Layout with Tables?

You should no longer use tables to create the layout of a website. This was done in the previous millennium. I only mention this here because you may have already looked or will look at one or the other source code of an older website, and there are still numerous websites from that time that use a table to lay out or align the document content. Most of the time, these are websites that aren’t maintained, or they come from developers who are no longer up to date. Today, you use CSS for the layout of a website.

5.1.4 Structuring Tables Using <thead>, <tbody>, and <tfoot>

As an alternative to the basic table elements of HTML you can also divide a table using the elements `<thead>`, `<tbody>`, and `<tfoot>` into a head, data, and foot section, respectively.

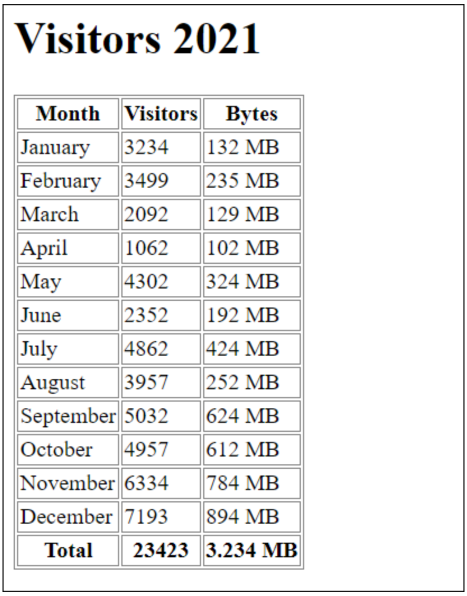
The table head is enclosed between `<thead>` and `</thead>` (`thead` = *table head*). It makes sense to use the `th` element for the individual cells. You can mark the actual data for the table using `<tbody>` and `</tbody>` (`tbody` = *table body*). If you want to write a range as a table foot, you must enclose it with `<tfoot>` and `</tfoot>` (`tfoot` = *table foot*).

Here's an example that uses these three elements in a table:

```
...
  <table>
  <thead>
    <tr>
      <th>Month</th>
      <th>Visitors</th>
      <th>Bytes</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th>Total</th>
      <th>23423</th>
      <th>3234 MB</th>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>January</td>
      <td>3234</td>
      <td>132 MB</td>
    </tr>
...
...
    <tr>
      <td>December</td>
      <td>7193</td>
      <td>894 MB</td>
    </tr>
  </tbody>
  </table>
...
```

Listing 5.4 /examples/chapter005/5_1_4/index.html

If you look at the HTML source code and the corresponding display in Figure 5.5, you'll notice that the web browser is able to reproduce the order of the table correctly on its own. Although the foot section was specified at the top in the source code, the web browser displays it appropriately at the bottom.



Visitors 2021

Month	Visitors	Bytes
January	3234	132 MB
February	3499	235 MB
March	2092	129 MB
April	1062	102 MB
May	4302	324 MB
June	2352	192 MB
July	4862	424 MB
August	3957	252 MB
September	5032	624 MB
October	4957	612 MB
November	6334	784 MB
December	7193	894 MB
Total	**23423**	**3.234 MB**

Figure 5.5 A Longer Table with <thead>, <tbody>, and <tfoot> Elements in Use

Dividing a table into three different sections is optional and usually doesn't affect the display in the web browser. This is a purely semantic representation. However, these elements are often used to format the appearance of these areas with CSS.

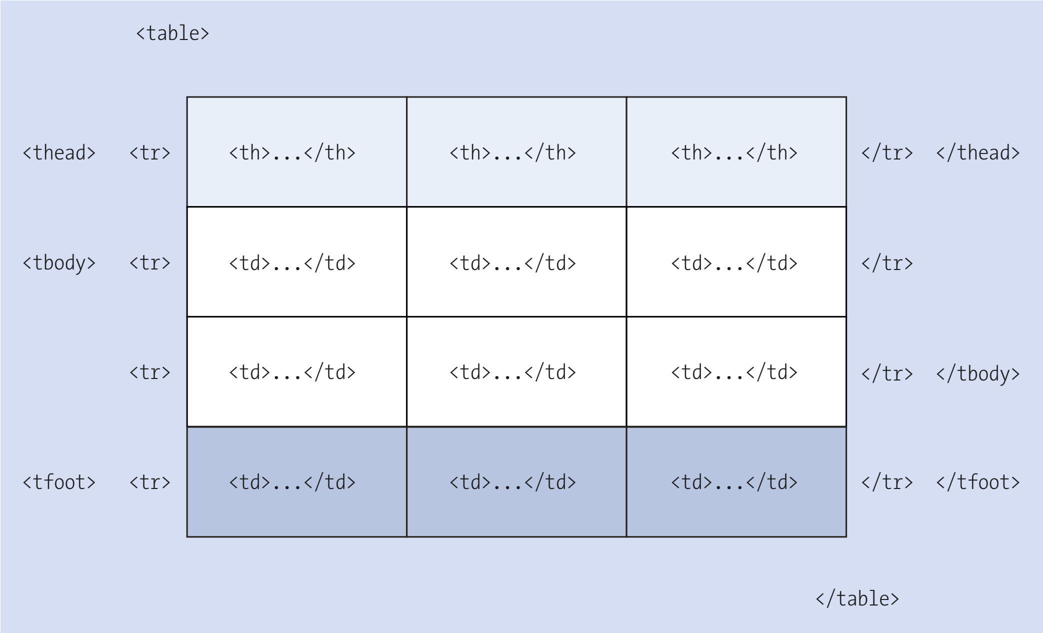


Figure 5.6 Only with CSS Can You Visualize These Sections Separately

In addition, when printing long tables across multiple pages, the web browser could use this division to print the table header and footer on each page as well. This makes it easier to see which column contains the individual data or what the data means. Another option is to scroll only the body area between `<tbody>` and `</tbody>` for long tables, while leaving the header and footer unchanged. Unfortunately, no web browser supports these features yet, but you may be able to do that yourself with CSS and JavaScript if necessary.

5.1.5 Grouping Columns of a Table Using <colgroup> and <col>

Just as you could divide the table rows into three sections using `<thead>`, `<tbody>`, and `<tfoot>`, you can also divide the individual columns into semantic and logical parts by means of the `<colgroup>` and `<col>` elements, if that made sense. Grouping columns is useful, for example, to apply specific CSS formatting to a particular column or group of columns, rather than repeating the style for each cell in the column.

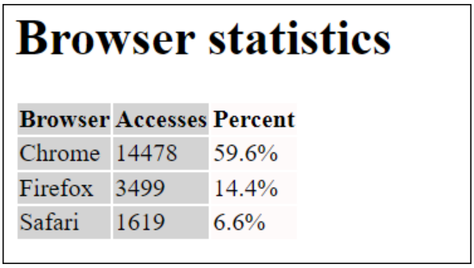
You need to write the `<colgroup>` and `<col>` elements after the opening `table` element and before any other elements such as `tr`, `thead`, `tfoot`, or `tbody`. You can open a column group using `<colgroup>` and close it with `</colgroup>` (`colgroup` = *column group*). To group a column, you can use the standalone `<col>` tag. If you want to combine several columns in one `col` element, you can do this using the attribute `span` and the number of columns as the attribute value.

Here's a simple example that illustrates what has just been described:

```
...
<table>
  <colgroup>
    <col span="2" style="background-color:lightgray;">
    <col style="background-color:snow;">
  </colgroup>
    <tr>
      <th>Browser</th>
      <th>Accesses</th>
      <th>Percent</th>
    </tr>
    <tr>
      <td>Chrome</td>
      <td>14478</td>
      <td>59.6%</td>
    </tr>
    ...
    ...
  </table>
...
```

Listing 5.5 /examples/chapter005/5_1_5/index.html

In Figure 5.7, the first two columns have been grouped together using `span="2"` and highlighted in color with CSS for demonstration purposes. The last column is a separate column group.



Browser statistics

Browser	Accesses	Percent
Chrome	14478	59.6%
Firefox	3499	14.4%
Safari	1619	6.6%

Figure 5.7 First Two Columns Have Been Grouped Together with Last Column as a Separate Column Group

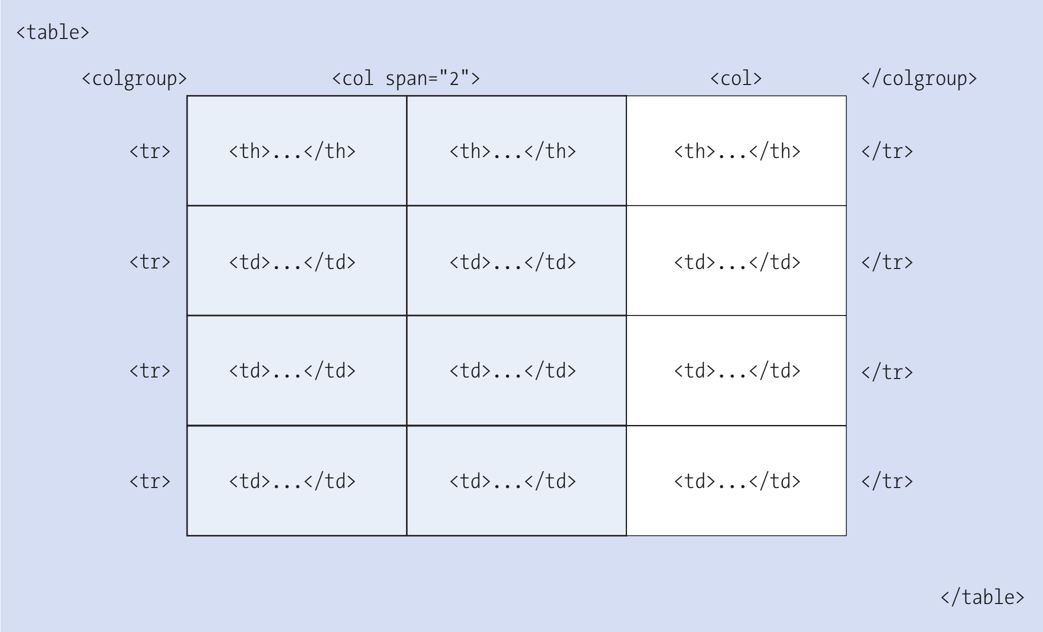


Figure 5.8 Semantic Division of Columns into Groups: Here, You Can See a Group with Two Columns and a Group with One Column

If, on the other hand, you want to use a separate group for each column, you can do this as follows:

```
  <table>
  <colgroup>
    <col style="background-color: lightgray;">
    <col style="background-color: snow;">
    <col style="background-color: lightgray;">
  </colgroup>
    <tr>
      <th>Browser</th>
      <th>Accesses</th>
      <th>Percent</th>
    </tr>
...
  </table>
...
```

Now each column has been grouped into its own `col` group. The advantage doesn't become apparent until you want to style columns with CSS. The semantic division into three columns can be found in Figure 5.9.

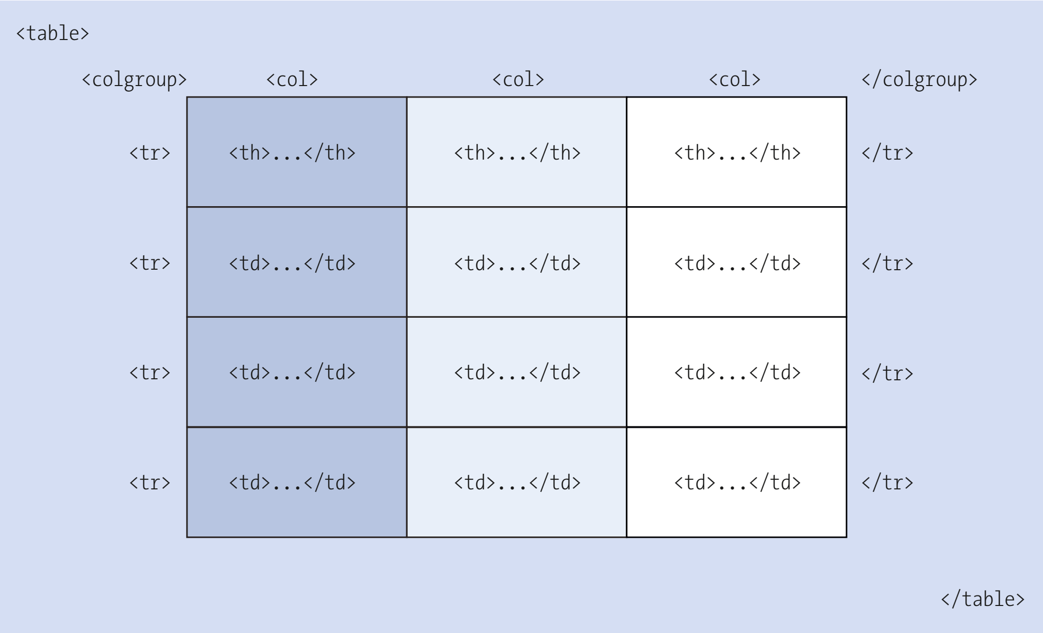


Figure 5.9 Semantic Division into Three Columns

5.1.6 Labeling Tables Using <caption> or <figcaption>

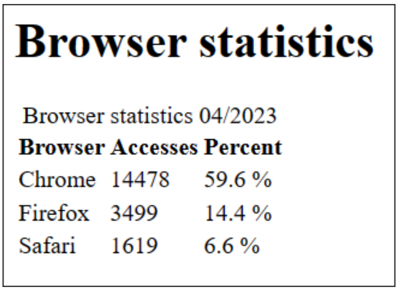
To assign a label to a table, you can either use the `caption` element, which must be used immediately after the opening `<table>` tag, or the new `figure` and `figcaption` elements.

Labeling a Table with <caption>

As mentioned previously, the `caption` element must follow immediately after the opening `<table>` tag. In addition, only one label can be used per table. Let's look at a simple example:

```
...
  <table>
    <caption>Browser statistics 11/2023</caption>
  <tr>
      <th>Browser</th>
      <th>Accesses</th>
      <th>Percent</th>
    </tr>
    <tr>
      <td>Chrome</td>
      <td>14478</td>
      <td>59.6%</td>
    </tr>
    ...
    ...
  </table>
...
```

Listing 5.6 /examples/chapter005/5_1_6/index.html



Browser statistics

Browser statistics 04/2023

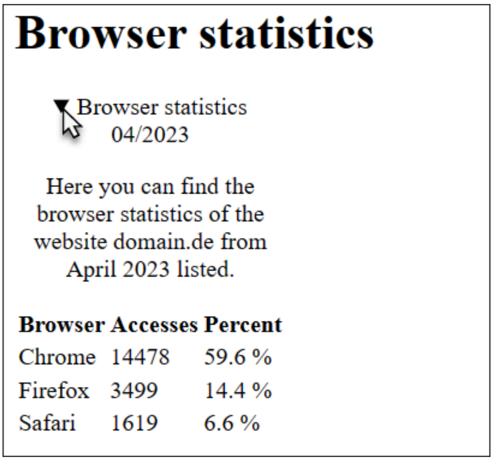
Browser	Accesses	Percent
Chrome	14478	59.6 %
Firefox	3499	14.4 %
Safari	1619	6.6 %

Figure 5.10 The Caption Is Displayed Centered above the Table by Default

Formatting <caption> with CSS

The web browsers usually display the caption centered above the table. With CSS, it's no problem to use the CSS features `text-align` and `caption-side` to align the table caption differently and position it somewhere else.

If you want to add notes to a table caption, you can place the HTML elements `details` and `summary` between `<caption>` and `</caption>`.



Browser statistics

▼Browser statistics 04/2023

Here you can find the browser statistics of the website domain.de from April 2023 listed.

Browser	Accesses	Percent
Chrome	14478	59.6 %
Firefox	3499	14.4 %
Safari	1619	6.6 %

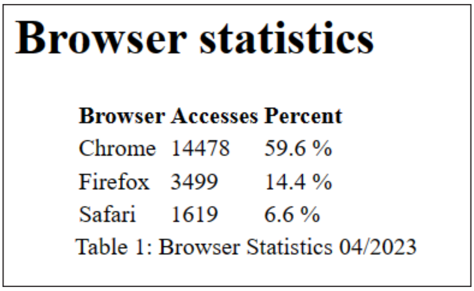
Figure 5.11 Expanding and Collapsing Information Thanks to the HTML Elements <details> and <summary> (Example in /examples/chapter005/5_1_6/index2.html)

Labeling a Table Using <figcaption>

I already described the `figcaption` and `figure` elements in Chapter 4, Section 4.2.9. It's a good idea to position tables between `<figure>` and `</figure>` and to insert a caption for this table at the beginning after the opening `<figure>` or at the end before the closing `</figure>`. Here's an example of how you can label a table using the new `figure` and `figcaption` elements:

```
...
  <article>
  <h1>Browser Statistics </h1>
  <figure>
  <table>
  <tr>
      <th>Browser</th>
      <th>Accesses</th>
      <th>Percent</th>
    </tr>
    <tr>
      <td>Chrome</td>
      <td>14478</td>
      <td>59.6%</td>
    </tr>
...
  </table>
  <figcaption>Table 1: Browser Statistics 04/2023</figcaption>
  </figure>
  </article>
...
```

Listing 5.7 /examples/chapter005/5_1_6/index3.html



Browser statistics

Browser	Accesses	Percent
Chrome	14478	59.6 %
Firefox	3499	14.4 %
Safari	1619	6.6 %

Table 1: Browser Statistics 04/2023

Figure 5.12 Labeling Tables Using <figure> and <figcaption>

5.2 Electronic References (Hyperlinks) Using <a>

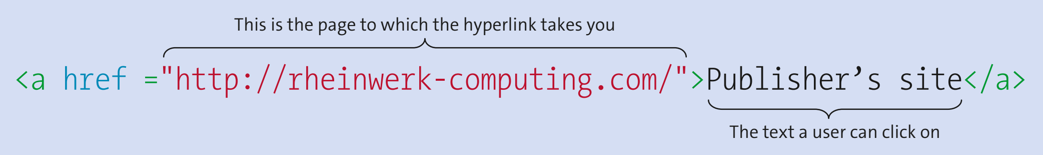
Hyperlinks are probably one of the most important elements of HTML because they make it possible to move from one website to another. You'll need hyperlinks, often just called links or references, to structure your project and implement references to other content. Starting from your main page, you often need links to other subpages and perhaps also links that return you to the main page. Only if you link multiple files does your website become a meaningfully usable website. Besides linking your own content, you can create links to other websites or other documents located elsewhere on the internet.

You can create a link in HTML using the `a` element (`a` = *anchor*). The text you write between `<a>` and `</a>` is called link text or reference text and is activated by using the `href` attribute in the opening `<a>` tag. The link text can be any text you want to write, but it isn't always helpful to simply write `Please click here`. A meaningful link text can help your visitors get where they want to go faster, as well as help visitors with screen readers. Elements other than text can be placed between `<a>` and `</a>`, such as a graphic as a link.

What Is Allowed between <a> and ?

As mentioned earlier, you can use other HTML elements besides text, such as graphics between `<a>` and `</a>`. You're even allowed to use grouping elements such as paragraphs, lists, articles, and block sets. As a matter of fact, you can use almost anything between `<a>` and `</a>`, apart from interactive elements such as links, form elements, and `audio` and `video`. That said, I recommend you don't put too much content into a single link between `<a>` and `</a>`. Screen readers will read the text aloud multiple times, and visitors might be overwhelmed by this because they are used to activating individual links in the traditional link style. Of course, this depends on the content of the website. I don't want to go into more detail here, but you now know that more HTML elements are available to you for links in HTML. If you've put an extreme amount of content between `<a>` and `</a>` and are no longer sure if it's still valid, you can validate the source code.

The most important attribute used with the `a` element is the `href` attribute. You can use the `href` attribute to specify the link users will be taken to when they click on the link text.



```
<a href ="http://rheinwerk-computing.com/">Publisher's site</a>
```

This is the page to which the hyperlink takes you
The text a user can click on

Figure 5.13 Classic Structure of a Hyperlink

A link text is commonly underlined by the web browser (usually in blue). If the link has already been visited, it will have a different color (usually purple). The color of links and visited links may vary depending on the web browser used, so there is no standard link color in this regard as each web browser has its own default stylesheet. You can change both at any time with CSS. Usually, when you move with the mouse pointer over the link, the cursor changes its shape into a hand, with the index finger pointing to the link. Most web browsers additionally display the address at the bottom left where the browser would land after a mouse click on the link.

If the link has been clicked, the web browser searches for the address (also called URL) specified in the link, loads it into the browser window, and replaces the old web page. If the address of the specified link can't be found, an error message will display, such as **404 - web page not found**. When the new web page has been loaded into the browser window, you can use the **Back** button to go back to the previous page.

Further Reading

I've already described the specification of terms; directory name; directory structure; and full, absolute, and relative paths in Chapter 3, Section 3.3. You can refer to that section if you have problems with the terms used in the following sections.

5.2.1 Inserting Links to Other HTML Documents on Your Own Website

When you create your website, these links are likely to be the first links you use to structure the loose collections of HTML documents into a coherent website—more precisely, to create the navigation of the website. If you want to provide a link to another page on the same website, you usually don't need to include the full domain name, but instead use a *relative URL*. The directory structure shown in Figure 5.14 should serve as an example.

The linking for the start page, *index.html*, to the other pages, *links.html*, *about.html*, and *legalnotes.html*, looks as follows:

```
...
  <nav>
    Blog |
    <a href="pages/links.html">Links</a> |
    <a href="pages/about.html">About me</a> |
    <a href="pages/legalnotes.html">Legal Notes</a>
  </nav>
  <h1>My Blog</h1>
  <p>Latest reports on HTML</p>
...
```

Listing 5.8 /examples/chapter005/5_2_1/index.html

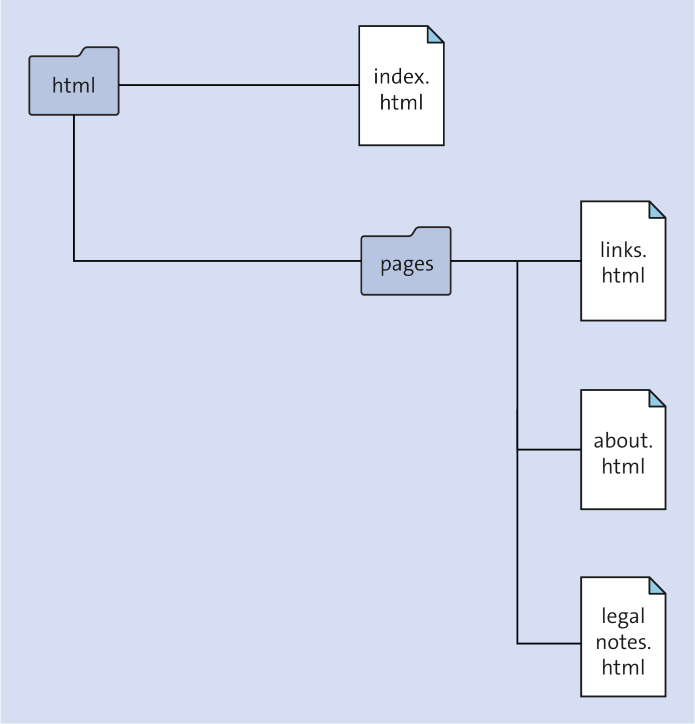
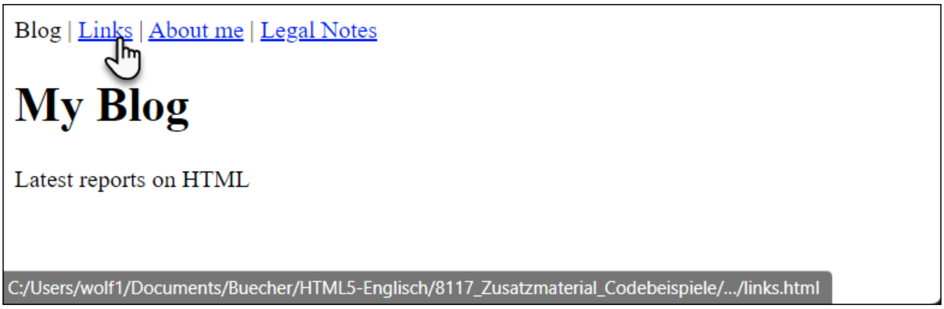


Figure 5.14 Directory Structure for an Example of Links to Other Pages on the Same Website



Blog | Links | About me | Legal Notes

My Blog

Latest reports on HTML

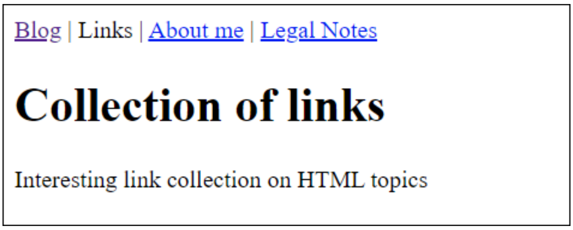
C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/.../links.html

Figure 5.15 Thanks to Linking via a Relative URL, Any Page Can Be Visited and Viewed within the Pages of the Same Website

Of course, you also need to adjust the links to the other pages, such as *links.html*, *about.html*, and *legalnotes.html*, in this example. When specifying the relative URL (see Figure 5.14), you must make sure that the pages (in this example) are located in a subfolder called *pages*. With regard to the *links.html* page, the specifications for the `href` attribute would look as follows:

```
...
  <nav>
    <a href="../index.html">Blog</a> |
    Links |
    <a href="about.html">About me</a> |
    <a href="legalnotes.html">Legal Notes</a>
  </nav>
...
```

Listing 5.9 /examples/chapter005/5_2_1/pages/links.html



Blog | Links | About me | Legal Notes

Collection of links

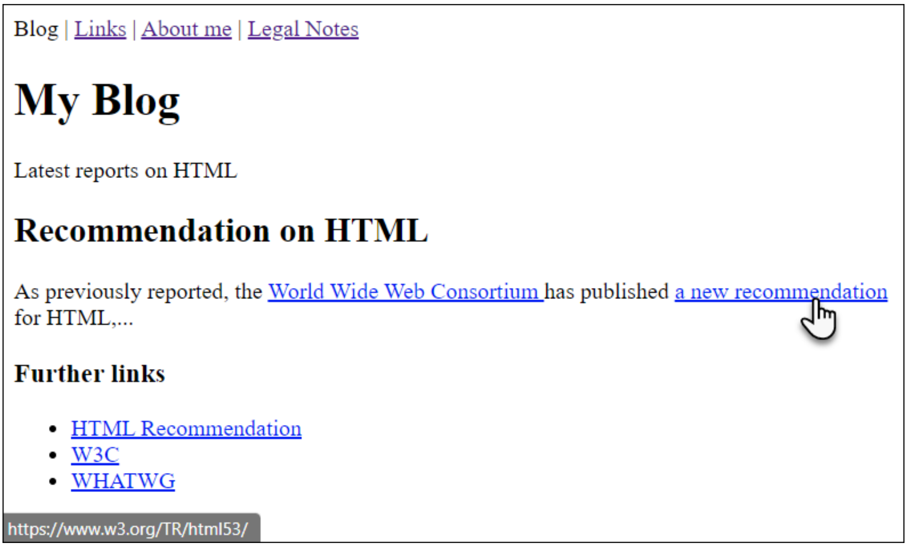
Interesting link collection on HTML topics

Figure 5.16 HTML Document links.html

Here you can see how to navigate from the *pages* subfolder (here *../index.html*) with *../* to the parent folder in which *index.html* is located. The other two files, *about.html* and *legalnotes.html*, are located in the same folder as *links.html*, so it's sufficient to specify only the file name. Both the *about.html* and *legalnotes.html* files need to be linked in the same way.

5.2.2 Inserting Links to Other Websites

Links to other websites must be written in the same way as the links to the pages of the same website, the only difference being that you must specify the complete address, that is, the *absolute URL*, to that page in the `href` attribute. For this purpose, here's a simple example in which links to external pages are included (see Listing 5.10).



Blog | Links | About me | Legal Notes

My Blog

Latest reports on HTML

Recommendation on HTML

As previously reported, the World Wide Web Consortium has published a new recommendation for HTML,...

Further links

- HTML Recommendation
- W3C
- WHATWG

https://www.w3.org/TR/html53/

Figure 5.17 Many Web Browsers Display the Link's Destination Address at the Bottom of the Status Bar When You Hover over It

```
...
  <article>
    <header>
      <h2>Recommendation on HTML</h2>
    </header>
    <p>As previously reported, the
       <a href="http://www.w3.org/">World Wide Web Consortium
       </a> has published
       <a href=" https://www.w3.org/TR/html53/">
       a new recommendation</a> for HTML,...
    </p>
    <aside>
    <h3>Further links</h3>
      <nav>
      <ul>
       <li>
       <a href="https://www.w3.org/TR/html53/">
        HTML Recommendation</a></li>
       <li><a href="http://www.w3.org/">W3C</a></li>
```

```
        <li><a href="http://www.whatwg.org/">WHATWG</a></li>
      </ul>
      </nav>
    </aside>
  </article>
...
```

Listing 5.10 /examples/chapter005/5_2_2/index.html

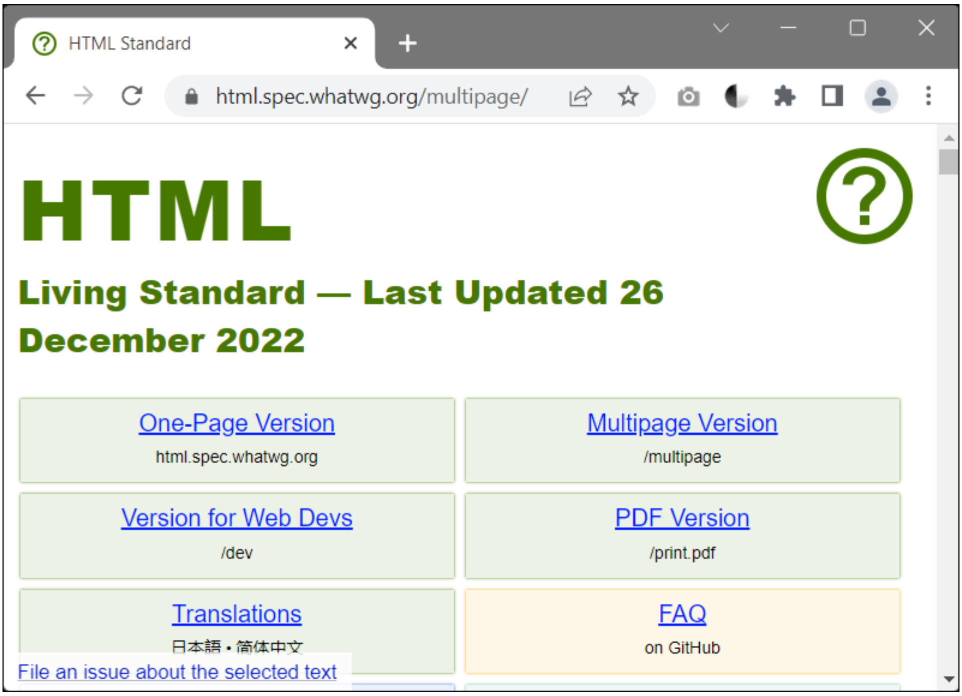

HTML Standard
html.spec.whatwg.org/multipage/
HTML
Living Standard — Last Updated 26 December 2022
One-Page Version
html.spec.whatwg.org
Multipage Version
/multipage
Version for Web Devs
/dev
PDF Version
/print.pdf
Translations
日本語 • 简体中文
FAQ
on GitHub
File an issue about the selected text

Figure 5.18 When You Activate the Link, the Destination Address Gets Loaded into the Web Browser and Displayed

5.2.3 Opening Links with the "target" Attribute in a New Window

You can use the HTML attribute `target` of the `a` element to open a reference target in a new window or tab. To do so, you only need to pass the attribute value `_blank` to `target`, for example:

```
<p>As previously reported, the
   <a target="_blank" href="http://www.w3.org/">W3C</a> has
   published a new draft for HTML,...
</p>
```

If you activated the link text `W3C` in this example, the target address (here, *www.w3.org*) is actually opened and loaded in a new window or tab. The primary goal of using `target="_blank"` is, of course, to avoid "losing" visitors to the page but to leave the page open so that they'll return to it when they have finished reading the page in the newly opened window or tab.

In addition to the most commonly used attribute value `_blank`, you can also use `_self` (= current window), `_parent` (= parent window), `_top` (= top window level), and names of windows that can be processed with JavaScript.

Using or Not Using the Attribute "target="_blank""?

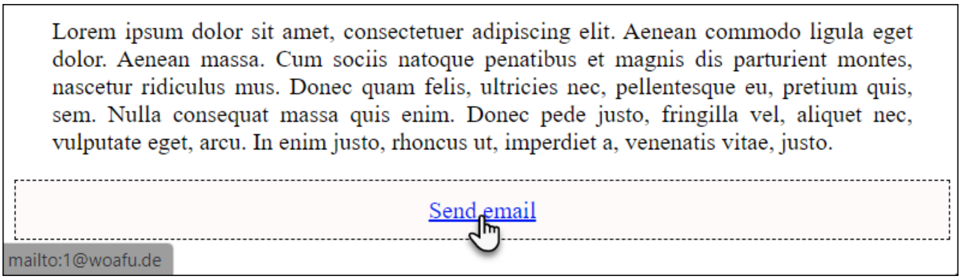
Even though some websites are quite fond of using this attribute, you shouldn't open a new window for every link, come hell or high water. In practice, you should leave it up to the user to decide whether or not to open a new page for a link. Even though you may be used to having countless tabs and multiple websites open at the same time, you should think about the more inexperienced visitors who just aren't or don't want to be that much into the World Wide Web. Use the `target="_blank"` attribute sparingly, and, if possible, let visitors know that a new window or tab will open when they activate the link.

5.2.4 Email Links with "href=mailto: . . ."

You certainly also know the sort of links where the email application opens with a specific email address when you activate it. These links are also created via the `a` element and the `href` attribute. Those email references start at `href` with `mailto:` and are followed by the desired email address, for example:

```
...
  <footer>
    <a href="mailto:1@woafu.de">Send email</a>
  </footer>
...
```

Listing 5.11 /examples/chapter005/5_2_4/index.html

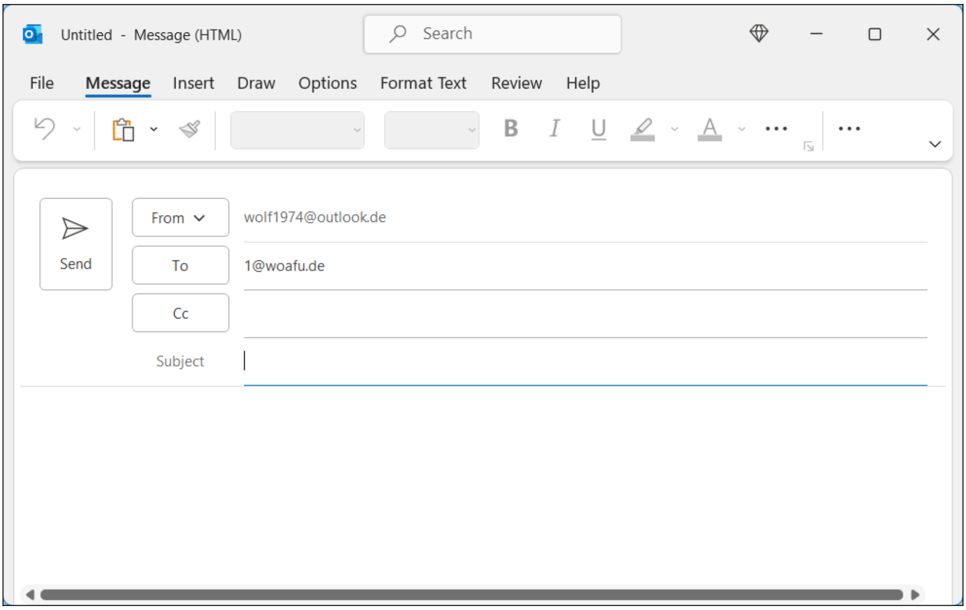


Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

Send email

mailto:1@woafu.de

Figure 5.19 When You Hover Your Mouse over the Link, You'll Usually See the Email Address Associated with That Link in the Status Bar



From: wolf1974@outlook.de

To: 1@woafu.de

Figure 5.20 When You Activate the Link, the Email Application often Opens, an Email Gets Created Automatically, and the Email Address Is Entered as the Recipient

The Functionality of a "mailto" Reference Isn't Reliable

Unfortunately, there is no guarantee that a `mailto` link will actually work. To make sure that it works, either the web browser must support the email creation and dispatch, or a local email application must be launched when a `mailto` reference is implemented. If a visitor doesn't use or hasn't set up a local email application, but only uses the classic webmail in the web browser, the `mailto` reference will only work if the web browser has been configured accordingly. In addition, there are web browsers that you can't configure in this way at all. It's therefore useful and recommended to also provide the email address in readable form, so that visitors who can't execute the mail reference can still send you an email.

Beware of Spam!

Due to the publication of email addresses on a website, you'll eventually have to face unsolicited commercial emails (spam) because there are web crawlers that scan websites for email addresses. You even have the obligation to name the email address in the legal notes. The only protection in this regard consists of avoiding mentioning the email address in the source code.

The first way to do that would be to include it as a graphic. However, this would be leave out people who rely on screen readers, and, furthermore, a "graphic email address" is also legally questionable. Often, obfuscation versions are still in use, in which the @ sign is replaced by *(at)* (*e.g., webmaster (at) donald-bear.com*). Likewise, the dot is written out as *(dot)* (e.g., *webmaster (at) donald-bear (dot) com*). Of course, this means the visitor has to enter the email address manually.

JavaScript obfuscation is another solution. There are many approaches to this. An interesting website with information about how you can hide your email address with JavaScript, for example, can be found at *http://alistapart.com/article/gracefulemailobfuscation*.

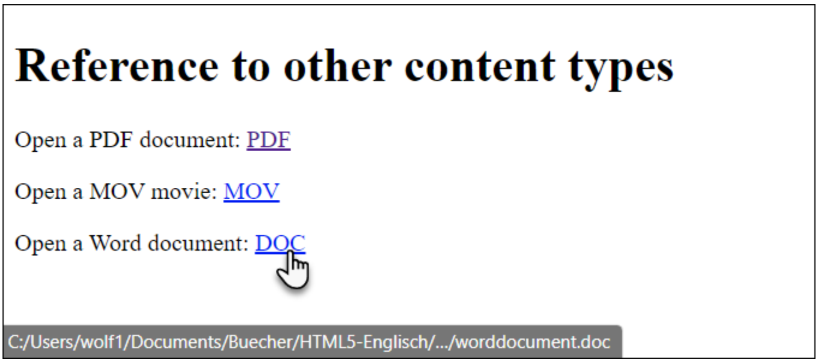
Google's *reCAPTCHA* module, which ensures that the email address gets displayed or forms are submitted by a human being, has also proven useful. However, this also entails extra work for the website visitor, who may have to type words or solve an image task. You can find more information on this topic, including an introductory video, at *http://google.com/recaptcha/intro/index.html*.

5.2.5 Setting Links to Other Types of Content

If you set links to other document types not commonly used on the web, such as Word, Excel, or PDF documents, it depends on the web browser to handle those document types further. As a web developer, you have no influence on this. Here, the general recommendation is first to use widely used formats. For example, a link to a PDF document is more likely to cause the web browser to launch a corresponding PDF reader and open the document within it than if the content type link is a platform-dependent or vendor-specific document (e.g., a Word document). Let's look at a simple example:

```
...
<h1>Reference to other content types</h1>
<p>Open a PDF document: <a href="document.pdf">PDF</a></p>
<p>Open a MOV movie: <a href="ganges.mov">MOV</a></p>
<p>Open a Word document: <a href="worddocument.doc"
   type="application/msword">DOC</a></p>
...
```

Listing 5.12 /examples/chapter005/5_2_5/index.html



Reference to other content types

Open a PDF document: PDF

Open a MOV movie: MOV

Open a Word document: DOC

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/.../worddocument.doc

Figure 5.21 Three Links to Different Types of Content

What will happen with the three links used in the example can't be completely predicted as it depends on the web browser. The PDF document shouldn't be a problem because the web browser should know how to handle it. It might be more difficult with the movie in MOV format, because that usually requires a QuickTime plug-in from Apple. Some web browsers offer the option to download and install the appropriate plug-in; others, however, don't.

The same applies to the Word document. If Word is installed on your computer, the web browser often provides a dialog box to open the document with Microsoft Word, or at least the option to select a corresponding application with which you want to open this document. Most of the time, however, only the option to download the document is provided. This again depends on how you've set the web browser.

Including the Content Type

For special types of content, you can provide the web browser with the multipurpose internet mail extension (MIME) type in the `type` attribute within the opening `<a>` tag, which is what I did in the example with `application/msword` for a Word document. The information is very useful for the web browser and also other web clients. It almost always makes sense to specify the file format if the link target isn't an HTML document.

Inform Visitors about What Is behind a Link

If you use non-HTML documents, you should definitely inform the visitor what is hiding behind the link and possibly how big the specific file is. You can use the global `title` attribute in the opening `a` element for this purpose, but it's recommended to mention more precise details directly with the link text. An example of how you don't want to do that looks like the following:

```
<a href="annualrevenue.pdf">Annual Revenue 2020</a>
```

The visitor will only see the link text `Annual Revenue 2020` here and may be confused about whether this link is to a PDF document that may take a little longer to load. For this reason, it's better to write the following:

```
<a title="Opens the PDF file with the annual revenue in 2020"
   href="annual-revenue.pdf">
   Annual Revenue 2020 (PDF, 3.9 MB)
</a>
```

5.2.6 Adding Download Links Using the "download" Attribute

You can also add links as download references irrespective of the content type (i.e., MIME type) of the link target. For this purpose, you want to use the `download` attribute in the opening `<a>` tag. Here, we use the same HTML code from example */examples/*

chapter005/5_2_5/index.html, but now all three files are provided for download using the `download` attribute:

```
...
<h1>Reference to other content types</h1>
<p>Download a PDF document:
   <a href="document.pdf" download>PDF</a></p>
<p>Download a MOV movie:
   <a href="ganges.mov" download="movie.mov">MOV</a></p>
<p>Download a Word document: <a href="worddocument.doc"
   download="worddocument.doc">DOC</a></p>
<p>Download an HTML document: <a href="website.html"
   download="website.html">HTML</a></p>
...
```

Listing 5.13 /examples/chapter005/5_2_6/index.html

The `download` attribute allows you to instruct a web browser to provide this file for download, even if it could display the file itself or knows the appropriate plug-in or add-on to do so, which it would usually use for such a content type.

You can use the `download` attribute as a standalone attribute, as shown in the first example with the PDF document. The name of the file that gets downloaded matches the specification in `href` (here, *document.pdf*). If the link in `href` doesn't contain a meaningful name, you can also assign a different name to the `download` attribute, as is the case in the example with the MOV movie whose actual document name is *ganges.mov*, but the download name of the file is *movie.mov*. The example with the HTML document is only intended to demonstrate that even typical web browser content types such as an HTML document with the attribute `download` are really only provided for download. Note, however, that this attribute only works if you try the example online.

Informing Visitors about What Gets Downloaded

As is the case with linking to non-HTML documents, you should let readers know what they are downloading and what they can use to view or reuse the document. For example, if you provide Excel spreadsheets with an annual revenue report for download, you should inform the reader what software they need to view the spreadsheet.

The same applies to ZIP archives. Here, too, you should add an additional note on how to unpack such an archive or a link to the relevant software. Keep in mind that many visitors don't know what to do with file extensions such as **.odt*, **.xls*, **.zip*, **.tar.bz*, and so on. You mustn't take this for granted just because you deal with countless data formats every day. It's recommended to include the file size when downloading. You could thus note the download of a large ZIP archive as follows:

```
...
<a title="Annual revenue in Excel format packed into a ZIP archive".
   href="archive.zip" download="annualrevenue2020.zip">
      Annual revenue 2020 (ZIP archive; 2.5 MB)</a>
   <small>(To unpack the ZIP archive, you need a
           packing program such as 7-Zip. The annual revenue figures are
           maintained in Excel format and thus require
           software that can view Excel spreadsheets.)
   </small>
...
```

Next to the `title` attribute, I've specified the file format (here, a ZIP archive) as well as the file size. In addition, I've written some small-print information between `<small>` and `</small>`.

5.2.7 Setting Links to Specific Parts of a Web Page

Nothing can be more annoying for visitors than reading a long scientific treatise of a specific topic on a web page and having to scroll up and down for a long time to get to a specific section. For those cases, you can set *anchors* with the global attribute `id`, which you can jump to via an ordinary link in the `a` element. For an example of such target anchors, view any Wikipedia page's table of contents of a topic. To link to a specific section of a web page, you only need the following:

- An anchor (jump marker) that you can create with the `id="anchorname"` attribute, for example:

  ```
  <h1 id="anchorname">Heading xyz</h1>
  ```

- A link pointing to the anchor via `href="#anchorname"`. For this purpose, the hash character # is written in front of the anchor name, for example:

  ```
  <a href="#anchorname">Go to heading xyz</a>
  ```

Here's a simple example of how you can set and use such jump markers in practice:

```
...
  <h1 id="top">Table of contents</h1>
  <ul>
    <li><a href="#intro">Introduction to HTML</a></li>
    <li><a href="#syntax">The Syntax of HTML</a></li>
    <li><a href="#versions">Versions of HTML</a></li>
    <li><a href="#techniques">Techniques around HTML</a></li>
    <li><a href="#practice">Getting Started</a></li>
  </ul>
```

```
  <h1 id="intro">Introduction to HTML</h1>
  <p>Lorem ipsum dolor sit amet ... <p>
  <p><a href="#top">To Table of Contents</a></p>
  <h2 id="syntax">The Syntax of HTML</h2>
  <p>Lorem ipsum dolor sit amet ... <p>
  <p><a href="#top">To Table of Contents</a></p>
  <h2 id="versions">Versions of HTML</h2>
  <p>Lorem ipsum dolor sit amet ... <p>
  <p><a href="#top">To Table of Contents</a></p>
  <h2 id="techniques">Techniques around HTML</h2>
  <p>Lorem ipsum dolor sit amet ... <p>
  <p><a href="#top">To Table of Contents</a></p>
  <h2 id="practice">Getting Started</h2>
  <p>Lorem ipsum dolor sit amet ... <p>
  <p><a href="#top">To Table of Contents</a></p>
...
```

Listing 5.14 /examples/chapter005/5_2_7/index.html

Figure 5.22 shows the example in the web browser. Thanks to jump markers, you can reach the desired section more quickly here.

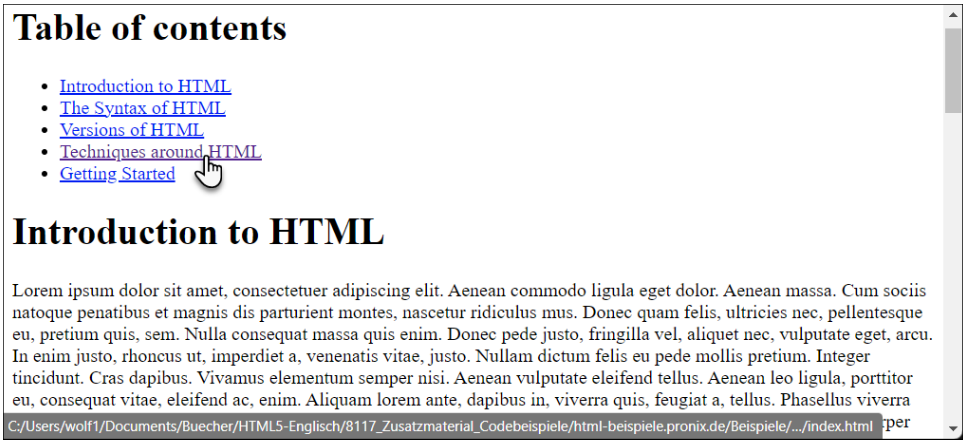


Table of contents

- Introduction to HTML
- The Syntax of HTML
- Versions of HTML
- Techniques around HTML
- Getting Started

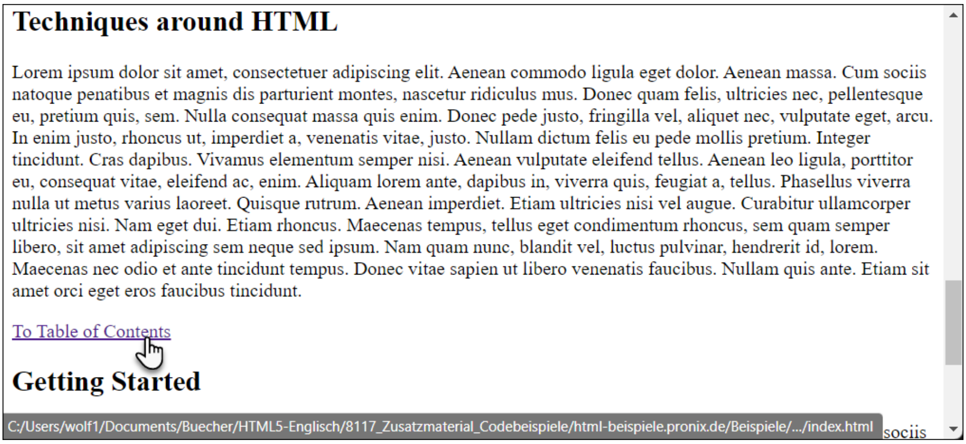
Introduction to HTML

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/html-beispiele.pronix.de/Beispiele/.../index.html per

Figure 5.22 Jump Markers Are Provided for Users to Reach Desired Sections Quickly

For example, if you activate the **Techniques around HTML** link, it will jump directly to the corresponding section with the jump marker, as you can see in Figure 5.23. Another link to jump back to the table of contents has also been added below each section.



Techniques around HTML

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed ipsum. Nam quam nunc, blandit vel, luctus pulvinar, hendrerit id, lorem. Maecenas nec odio et ante tincidunt tempus. Donec vitae sapien ut libero venenatis faucibus. Nullam quis ante. Etiam sit amet orci eget eros faucibus tincidunt.

To Table of Contents

Getting Started

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/html-beispiele.pronix.de/Beispiele/.../index.html sociis

Figure 5.23 Clicking the "Techniques around HTML" Link Jumps the User Directly to the Corresponding Section

Setting Anchors Using the "id" Attribute

Before you can create a link to a specific part within a web page, you must set the jump marker (or an anchor) using the global `id` attribute (`id="anchorname"`) within an opening HTML tag. In the example, this was done for the main headings `<h1>` and `<h2>` (e.g., `<h2 id="techniques">`). The attribute value of `id` must start with a letter or an underscore (never a number) and mustn't contain any spaces. It's also advisable to use a descriptive name to avoid losing track. You shouldn't use meaningless designations such as `anchor1`, `anchor2`, and so on. Also note that this is case-sensitive.

Referencing an Anchor Using "#anchorname"

To use a link to the anchors, you need to specify the attribute value for the anchor in `href` in the opening `<a>` tag. For example, if the anchor is `<h2 id="techniques">`, you want to write the hash character `#` in front of the anchor name (here, `"techniques"`). With reference to our example, you'd have to write this as follows:

```
<li><a href="#techniques">Techniques around HTML</a></li>
```

If you activate this link, the HTML document will jump to the element where the value of the `id` attribute is `"techniques"`. In this example, that would be the `h2` element with the heading *Techniques around HTML*.

Creating Links to a Specific Section of Another Website

Likewise, you can create a link to a section of another HTML document. This requires that the other HTML document contains a corresponding anchor. If the anchor is in another document, you can create a reference to it as follows:

```
<a href="tech.html#techniques">Techniques around HTML</a>
```

This would cause a jump to the section with the `#techniques` anchor in another HTML document located in the same directory and whose filename is *tech.html*.

If the file with the anchor is even on another website, you must specify the complete URL there:

```
<a href="http://www.domain.com/path/tech.html#techniques">...</a>
```

Of course, it's also possible to use a link to parts of third-party websites. However, it goes without saying that you can't set an anchor here, but only link existing anchors. For example, here's a link to an anchored section of a Wikipedia page:

```
<a href="https://en.wikipedia.org/wiki/HTML#Versions">...</a>
```

Here, you'd jump directly to the Wikipedia page with the entry `HTML` to the `#Versions` anchor. This is based on the assumption that the anchor exists—which was still the case when the book went to press, but could change at any time. If the anchor no longer exists or is wrong, the website will be called, and the anchor will get ignored as if `#anchorname` hadn't been specified when linking to the `a` element.

5.2.8 Creating Links to Phone Numbers

Because more and more users mostly go online with a mobile device, you can also create a link to a phone number. When the user taps on it, this number can be called directly from the website. However, the respective web browser must also support this function. This is shown in the following example:

```
...
<p>
  Customer Service Rheinwerk Publishing:
  <a href="tel:+1.781.228.5070.200">+1.781.228.5070 Ext. 200</a>
</p>
...
```

Listing 5.15 /examples/chapter005/5_2_8/index.html

The phone reference starts with `tel:` followed by the number. It's recommended to use the plus sign and the country code and to omit the leading 0 in the area code. Spaces can be written as a period. Because it depends on the web browser whether or not the number gets dialed immediately, you should also list the phone number.

You can also set other services such as Skype as a link, which allows you to start a Skype session with one click. The prerequisite for this is, of course, that the user also uses Skype. A Skype example is shown here:

```
...
<p>
  Start a Skype call:
  <a href="skype:pronix74">Skype: pronix74</a>
</p>
...
```

Listing 5.16 /examples/chapter005/5_2_8/index.html

Again, there's no guarantee that the web browser will start a Skype session, so you should also list the appropriate data. The same applies to FaceTime.

Automatic "tel:"

There are mobile web browsers, such as Safari, which automatically recognize a phone number and generate a *tel: link* from it. This is convenient, but perhaps not always desirable, especially when the number in question isn't a phone number at all. There are `meta` tags available for this, which you can use to instruct the web browser not to use this feature on the web page.

For Safari:

```
<meta name="format-detection" content="telephone=no">
```

For BlackBerry:

```
<meta http-equiv="x-rim-auto-match" content="none">
```

5.2.9 HTML Attributes for the HTML Element <a>

Finally, the HTML attributes for the links should be explained here, which can be quite useful for search engines, among other things. In Table 5.2, you can see an overview of all existing attributes for the `a` element. You already know some of them.

Attribute	Description
`download`	This attribute indicates that you provide the referenced target for download, even if the web browser could display the target's content type by itself.
`href`	This attribute specifies the URL of the page the link will lead to when activated.
`hreflang`	This attribute specifies the language of the linked document. The usual language abbreviations are permitted as specifications (e.g., `de` for Germany).
`media`	This attribute allows you to specify information about the media for which the link target has been optimized. You can either enumerate media types, separate by commas, or specify `all` for all media types.

Table 5.2 Attributes for Links with the <a> Element

Attribute	Description
`rel`	You already know the attribute from the `link` element described in Chapter 3, Section 3.5.1 , which you can refer to if you need more information. This attribute allows you to determine the type of linking. Especially for the `a` element, the `rel` attribute values `bookmark`, `external`, `nofollow`, and `noreferrer` are of special importance because they can only be used in the `a` element: ■ `rel="bookmark"`: This value enables you to specify that the link target is a parent section (or document) of the current document. It effectively represents a link back to an extensive HTML document, as is the case with scientific or technical documents. In practice, this link type is also used for permalinks so that visitors can view an older version of the current document. ■ `rel="external"`: This value indicates that the link belongs to an external website. It often happens that this attribute is separately formatted with CSS. ■ `rel="nofollow"`: This value allows you to instruct the web crawlers not to follow the link. ■ `rel="noreferrer"`: This value instructs the visitor's web browser not to use a referrer address when clicking on the link, which should prevent the destination web server from receiving information about where the visitor came from. However, you can't use the attribute values `icon`, `pingback`, `prefetch`, and `stylesheet` for `a` elements.
`target`	This attribute enables you to enter where the link target should be opened. Possible values are listed here: ■ `_blank`: New window/tab. ■ `_parent`: Parent window. ■ `_self`: Current window. ■ `_top`: Top window level. ■ `framename`: Name of the window opened with JavaScript and also assigned in it.
`type`	This attribute allows you to inform the web browser about the MIME type (file format) to which the linked file belongs. This specification is useful if the target isn't an HTML document.

Table 5.2 Attributes for Links with the <a> Element (Cont.)

5.3 Summary

In this chapter, you've learned about some essential HTML elements. The most important elements you'll probably find and use on almost every website are the following:

- The `a` element, which allows you to create hyperlinks
- Tables that let you present related data and information in a grid of rows and columns

Chapter 6

Graphics and Multimedia

With HTML, you can add graphics, videos, and other multimedia content to the HTML document.

The options to add multimedia content, such as graphics, animations, video, or audio, to a website have become more versatile and easier in HTML. For this reason, this chapter deals with the following topics:

- **Images**
  Today, it's hard to imagine a website without images, graphics, or logos, so here you'll learn how to add images to an HTML document.
- **Link-sensitive graphics**
  You'll learn how to embed multiple hypertext links within a graphic.
- **Flexible images**
  You can also load the appropriate image from multiple image sources with HTML.
- **Favorites icons**
  Everyone knows those little icons in the address bar, tab, or bookmarks. Here, I'll explain how you can add such a *favicon* to a website.
- **Vector graphics**
  If you want to use vector graphics for your website, you can read here what options are available to you to do so.
- **Drawing graphics**
  With HTML, you can also draw something directly on a web page. However, HTML only provides the canvas for this, which you'll learn how to create.
- **Video**
  Playing videos is also an issue and has become much easier thanks to the `video` element. You'll get to know which video formats are supported. Likewise, you'll learn how you can add a video to your website via YouTube.
- **Audio**
  Playing music and sounds on web pages has also been simplified significantly thanks to the `audio` element. You'll learn how to insert your own audio files into an HTML document.
- **Other active content**
  There's a lot more other content, such as PDF documents, Flash animations, and Java

applets, for which there's no special HTML element available. But I'll also describe how you can include such active content in an HTML document.

Table 6.1 lists the HTML elements used with graphics and multimedia.

HTML Element	Meaning
`<img>`	Including a graphic file in an HTML document
`<map>`, `<area>`	Creating a link-sensitive graphic
`<picture>`, `<source>`	Loading the appropriate image from several image sources
`<svg>`	Integrating a scalable vector graphic into the web page
`<math>`	Including a formula written with MathML
`<canvas>`	Providing a canvas for drawing graphics
`<video>`	Playing video files without plug-ins
`<audio>`	Playing audio files without plug-ins

`<embed>`
`<object>`
`<iframe>`
	Embedding active elements such as PDF documents, Flash animations, Java applets, Word documents, and many others

Table 6.1 Brief Overview of the HTML Elements for Graphics and Multimedia Covered in This Chapter

6.1 Embedding Images Using <img>

The internet without graphics and images is hardly imaginable today. There are several types of images that you can add to an HTML document. These include logos, charts, photos, illustrations, animations, and advertising banners, to name just a few examples. You can embed such images using the standalone `img` element.

6.1.1 Adding Images to an HTML Document

You can add images to a web page via the standalone `img` element (`img` = *image*). The graphic gets inserted at the position where you write the `img` element in the HTML document. In addition, no line break is used after the graphic if you use a graphic in the body text.

At least the HTML attributes `src` and `alt` must be present in the `<img>` tag. You can use the `src` attribute to specify where the web browser finds the image file. The `alt` attribute, on the other hand, is used for an alternative description of the image in case the image can't be loaded or displayed, and it also helps people with visual impairments because the text it contains is read aloud by screen readers (and should describe what is shown in the image). While this alternative text can be of almost any length, you should limit the text to the essentials. A recommended limit is about 12 to 16 words or 75 to 125 characters.

Does the “alt” Attribute Necessarily Have to Be Used in the <img> Element?

You don’t necessarily have to use the `alt` attribute in the `img` element, but if the attribute is missing, the validation check will report an error asking you to use it. This is confusing at first because it says that the `alt` attribute must be used, whereas the attribute value isn’t mandatory. If you don't want to use the `alt` attribute, then at least the surrounding text should describe the image. For example, if there’s no relevant content in a graphic (e.g., an empty graphic as a placeholder), and a description with the `alt` attribute makes no sense at all, it’s recommended to use an empty `alt=""`.

Description Text in “alt” for Search Engines

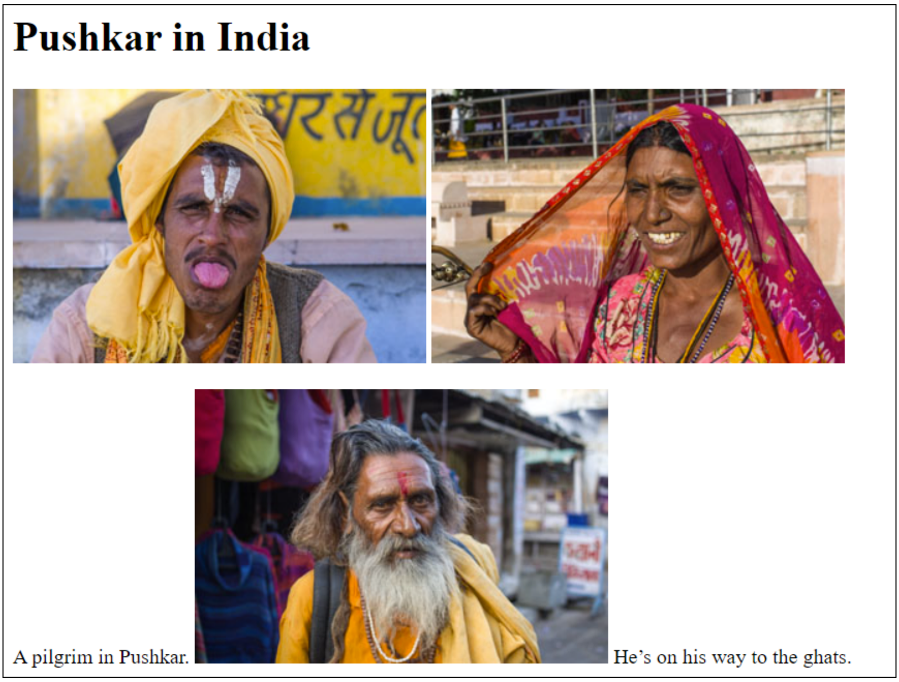
The `alt` attribute is also important for search engines because a web crawler can’t “see” what’s in an image. Thus, web crawlers have to rely on a meaningful description with meaningful keywords in the `alt` attribute. While there’s no official recommendation on how much you should put into the `alt` attribute, you shouldn’t overdo it here. However, the more data the web crawler has at its disposal, the more positive the effect will be on the website’s ranking. Thus, in addition to a meaningful `alt` attribute, it’s advisable to provide a short description of the image in the surrounding text.

The following example demonstrates the `img` element in use:

```
...
  <h1>Pushkar in India</h1>
  <p>
    <img src="images/TongueOut.jpg" alt="Pilgrim in Pushkar">
    <img src="images/singer.jpg" alt="Singer in Pushkar">
  </p>
  <p>
    A pilgrim in Pushkar. <img src="images/pilgrim.jpg"
    alt="Pilgrim in Pushkar"> He’s on his way to the ghats.
  </p>
...
```

Listing 6.1 /examples/chapter006/6_1_1/index.html

As you can see in Figure 6.1, images are left-aligned by default to match the alignment of the text. The top two images are displayed side by side if they fit the width of the browser window. This figure also shows how to align an image between body text flush with the font baseline.



Pushkar in India

A pilgrim in Pushkar.

He's on his way to the ghats.

Figure 6.1 Three Images Were Added to an HTML Document Using the <img> Element

Using a Custom Folder for Images

For creating a website, it has proven useful to store all images in a separate directory. Especially when the website becomes more extensive, you can maintain a better overview of where the individual files are stored. Graphics and images are often placed in a directory named *images* or *pictures*. When you want to insert a graphic into a web page, you'll be grateful to simply use `src="/images/imagename.jpg"` or similar, instead of also having to bother about where the graphic is located and what the reference to the image is.

Using the Global "title" Attribute for Images

The global `title` attribute is often used for images. The content of the `title` attribute gets displayed when you hold the mouse cursor over the image. The following figure shows an example with the `title` attribute.

Using the `title` attribute to provide additional information about images is fine here. If you use an image as a link to another page, it would make more sense to use this `title` element to indicate what happens when the user clicks on this image. For example, if an enlarged version of the image displays in a new window, the following use of the `title` attribute would make sense semantically:

```
...
<a href="images/TongueOut_big.jpg" target="_blank">
   <img src="images/TongueOut.jpg" alt="Pilgrim in Pushkar"
   title="Opens this image in a larger version in a
   new window or tab.">
</a>
...
```

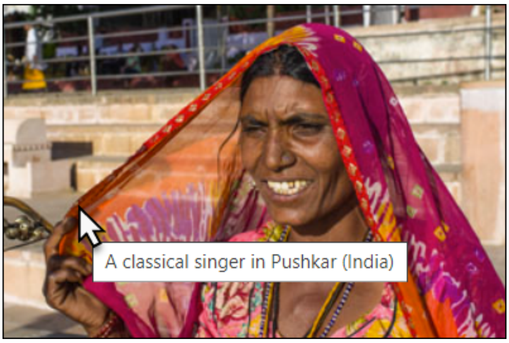

A classical singer in Pushkar (India)

Figure 6.2 This Additional Piece of Information Was Added in the <img> Tag with “title="A classical singer in Pushkar (India)"”

As you can see in Figure 6.3, the image has been captioned with a link to a larger version of the image. Accordingly, the description with the `title` attribute was changed.

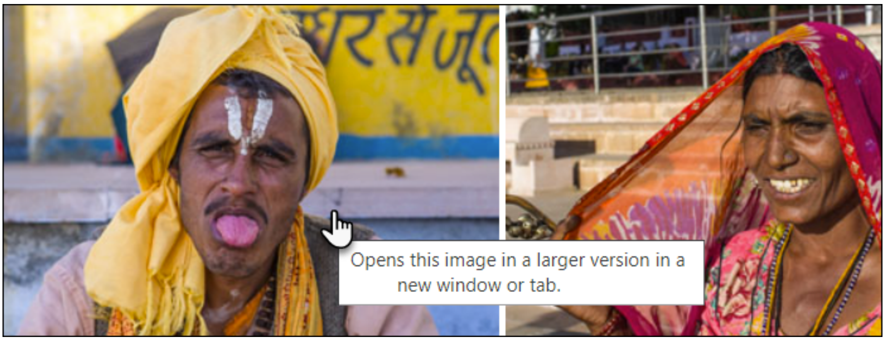


Figure 6.3 The “title” Attribute Allows You to Indicate That the Image Is Available in a Larger Version, Which You Can Open via the Link

Adding an Image or Linking an Image to a Website?

Note that an image isn’t technically inserted into the HTML document; instead, it’s linked to the HTML document. The `<img>` tag only creates the necessary space for the referenced content.

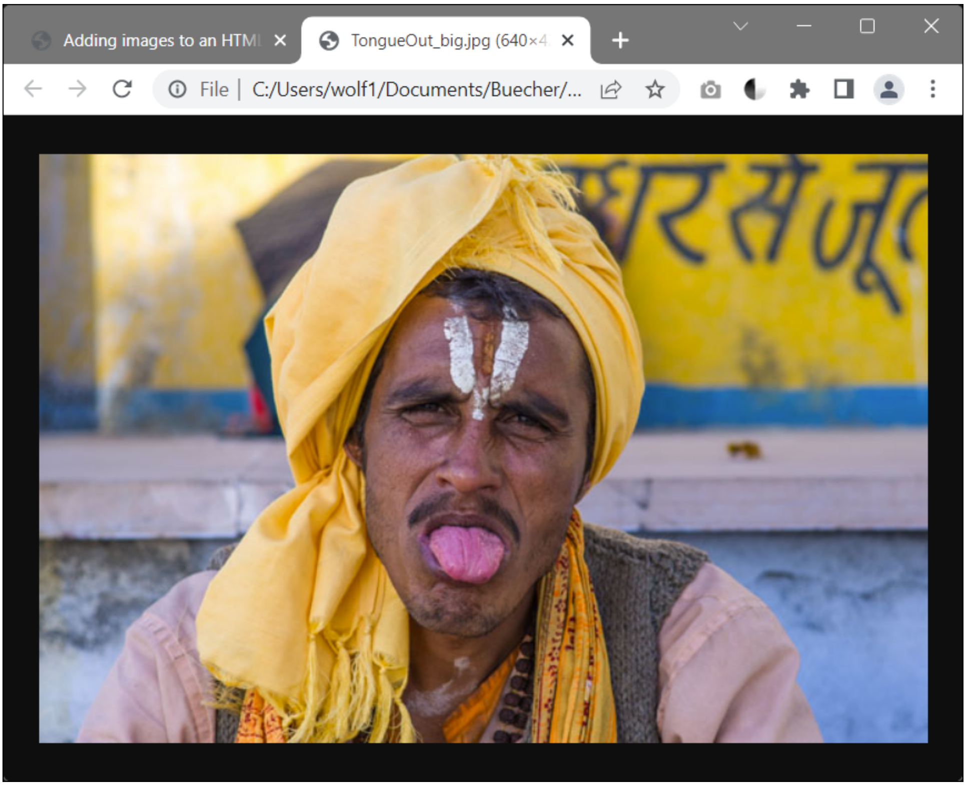


Figure 6.4 When the Visitor Clicks on the Image with the Link, the Larger Image Will Get Displayed in a New Tab

6.1.2 Specifying the Height and Width of a Graphic

Other useful attributes are `height` and `width`, which you can use to specify the height and width (in pixels) for the graphic. The purpose of using this information is that loading images usually takes longer than loading the HTML code to be displayed. This way, the web browser can already build and display the complete HTML document, leaving enough space for the images right away—more precisely, a free space is displayed for the images that haven't been loaded yet. This avoids unsightly subsequent corrections and rescreening with the graphics. Visitors with a fast internet connection won't even notice this loading process. However, if the internet connection is slow, a mobile device is being used, or the page is simply taking longer than usual to load, reading the content this way while the images are still loading is easier because the entire content doesn't get repositioned with each reloaded image.

Here's an example with the attributes `height` and `width`:

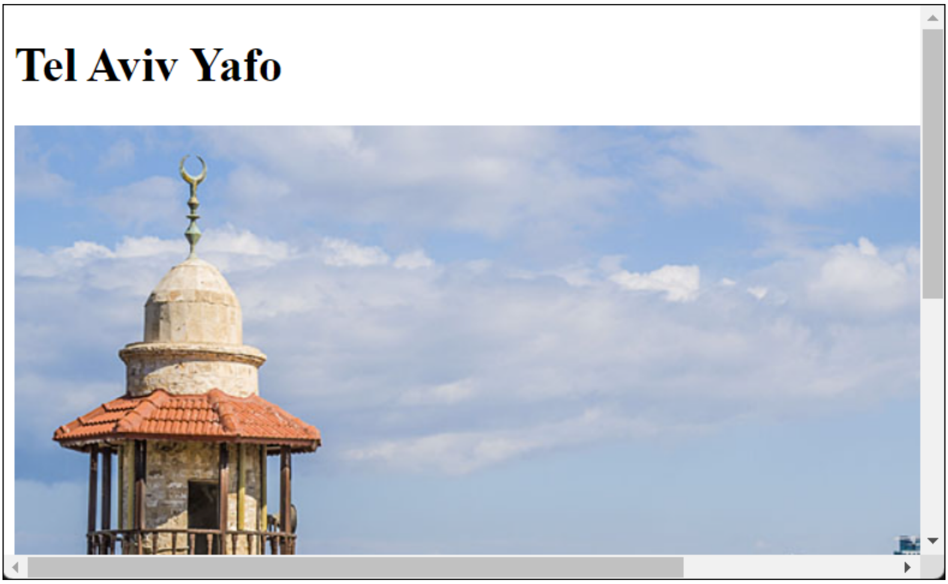
```
...
  <p>
    <img src="images/TongueOut.jpg" alt="Pilgrim in Pushkar"
```

```
     width="320" height="212">
    <img src="images/singer.jpg" alt="Singer in Pushkar"
     width="320" height="212">
  </p>
  <p>
    A pilgrim in Pushkar. <img src="images/pilgrim.jpg"
    alt="Pilgrim in Pushkar" width="320" height="212"> He is on
    his way to the Ghats.
  </p>
...
```

Listing 6.2 /examples/chapter006/6_1_2/index.html

Scaling Images in the Web Browser Using “height” and “width”

You can also scale the display size of images in the web browser using the `height` and `width` specifications. This “trick” of scaling images in the web browser is often used when an image is a little too large, as shown in Figure 6.5.



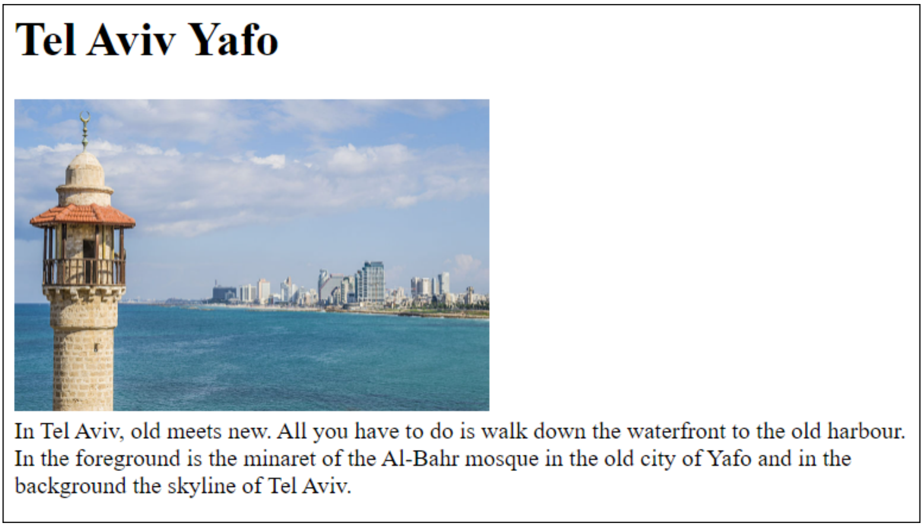
Tel Aviv Yafo

Figure 6.5 The Image Is Too Large in Its Original Size of 800 × 526 Pixels to Be Displayed Appropriately in the Window

You could scale this image down in the web browser using `height` and `width` and by specifying the desired dimensions:

```
...
<img src="images/yafo.jpg" alt="Tel Aviv Yafo"
  width="320" height="210">
...
```

The original image with 800 × 526 pixels is scaled down to 320 × 210 pixels by specifying `width` and `height`. The scaling process only takes place in the web browser's display and doesn't change the file size itself. Alternatively, you can specify only the width or the height with reduced values and let the web browser adjust the other value proportionally. You can see the result of scaling down in the web browser from 800 × 526 to 320 × 210 pixels in Figure 6.6, where the photo from Figure 6.5 is displayed smaller. Loading the file takes the same amount of time as before because the file size is the same. You can find the example in */examples/chapter006/6_1_2/index2.html*.



Tel Aviv Yafo

In Tel Aviv, old meets new. All you have to do is walk down the waterfront to the old harbour. In the foreground is the minaret of the Al-Bahr mosque in the old city of Yafo and in the background the skyline of Tel Aviv.

Figure 6.6 The Image Was Scaled Down by the Web Browser Using the "width" and "height" Attributes

Scaling in the Web Browser versus Using an Image Editor

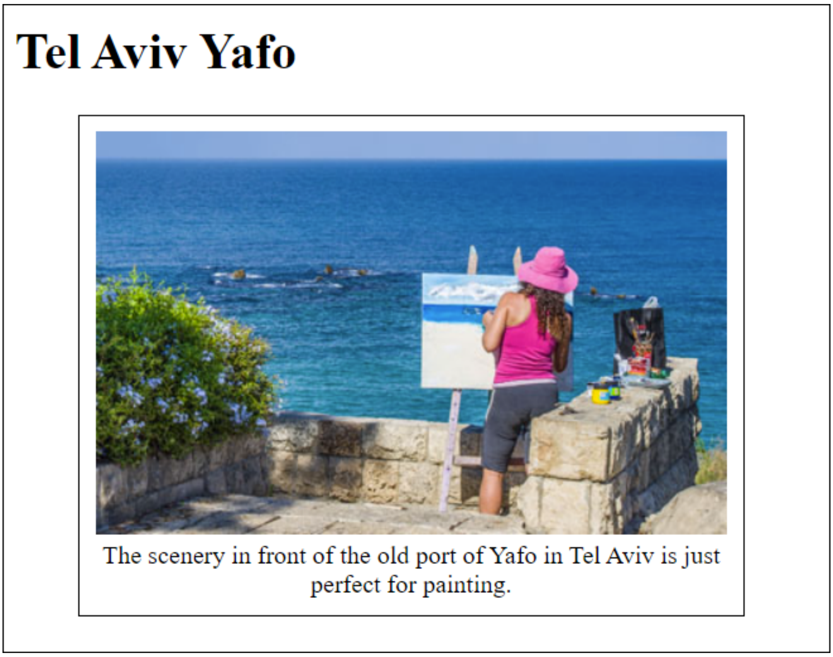
While scaling in the web browser using `width` and `height` is possible and allowed, you should use an image editor in practice to scale the images to the actual size if possible for two reasons: you reduce the amount of data (traffic), and you don't run the risk of ugly and rough shrinkage of the graphic. After all, you're relying on your visitors' web browsers to render the site, giving you a bit of control over how the images are displayed.

6.1.3 Labeling Images Using <figure> and <figcaption>

The two HTML elements `<figure>` and `<figcaption>` (described in Chapter 4, Section 4.2.9) are well suited for adding a caption to a graphic. To do this, you just need to enclose the image, that is, the `img` element, between `<figure>` and `</figure>` and then add the `figcaption` element with the caption. If you write `<figcaption>... </figcaption>` right after the opening `<figure>`, the caption will appear above the image. If you write `<figcaption>...</figcaption>` directly before the closing `</figure>`, the caption will appear below the image. Figure 6.7 shows the following simple example in execution:

```
...
<figure>
  <img src="images/painter.jpg" alt="Painter in Tel Aviv"
   width="400" height="256"><br>
   <figcaption>The scenery in front of the old port of Yofa in
    Tel Aviv is just perfect for painting.</figcaption>
</figure>
...
```

Listing 6.3 /examples/chapter006/6_1_3/index.html



Tel Aviv Yafo

The scenery in front of the old port of Yafo in Tel Aviv is just perfect for painting.

Figure 6.7 The Caption Has Been Formatted with CSS for Clarity, So That the <figure> Element Can Be Seen More Clearly

Merging Multiple Images

You can also use the `figure` element to include multiple images or different content. No matter how many elements you specify between `<figure>` and `</figure>`, you can use only one `figcaption` element for a caption. In addition, `figcaption` must be the first or the last element of the `figure` element.

6.1.4 HTML Attributes for the HTML Element <img>

Finally, the following table describes the attributes of the `img` element, the most important of which you've already learned about.

Attribute	Description
`alt`	You can specify alternative text to be displayed when images or graphics can't be displayed. This attribute is also helpful for visitors with a visual impairment or for search engines.
`height`	You can specify the vertical extent (height) of the image in pixels.
`ismap`	You can use this Boolean value if the images are a server-side image map. This server-side technique is rarely used, so I won't go into the detail here. I'll describe link-sensitive graphics (client-side technique) in Section 6.2.
`src`	You can specify the link destination to the image file. In addition to the file name, you can specify a relative or absolute path as well as an entire internet address. The specification of this attribute is mandatory.
`usemap`	You can specify the name of an image map to be associated with the image. This is a client-side link-sensitive graphic, which will be described in Section 6.2.
`width`	You can specify the size in the horizontal direction (width) of the image in pixels.

Table 6.2 The HTML Attributes of the <img> Element

6.2 Creating Link-Sensitive Graphics (Image Maps)

Link-sensitive graphics (also called *image maps*) are just links embedded within a graphic. Such an image map defined in HTML consists of three parts:

- **Image**
  This is the actual image that's added to the HTML document with the `img` element. Additionally, within the `<img>` tag, you must specify the HTML attribute `usemap` with an anchor to a `map` element:

  ```
  <img src="image.jpg" alt="alternText" usemap="#mapname">
  ```

- **`map` element**
  You also need the `map` element with the anchor name you specified earlier in the `img` element with the `usemap` attribute. The anchor name in the `map` element must be specified without the leading `#` in the name, unlike the `img` element. Between the introductory `<map name="mapname">` and the closing `</map>`, you can define the link-sensitive sections for the graphic. It's up to you to decide where exactly you write the `map` element in the HTML document. Thus, you can specify the `map` element before

the `img` element, for example. In practice, it's recommended to position the `map` element at the beginning or end of the document body.

- **Coordinates**
  You need the coordinates with the actual link-sensitive area for the graphic. For this purpose, a separate `area` element is used between `<map>` and `</map>` for each area. Within the `<area>` tag, you can define square areas, circular areas, or areas with any number of corners (polygon). For each `area` element, you define a "clickable" area in the image.

HTML Element	Meaning
`<img>`	Inserting a graphic with an anchor in a `map` element
`<map>`	Defining an area for the link-sensitive graphic
`<area>`	Defining a clickable area in the image

Table 6.3 Overview of the Necessary Elements for Link-Sensitive Graphics (Image Maps)

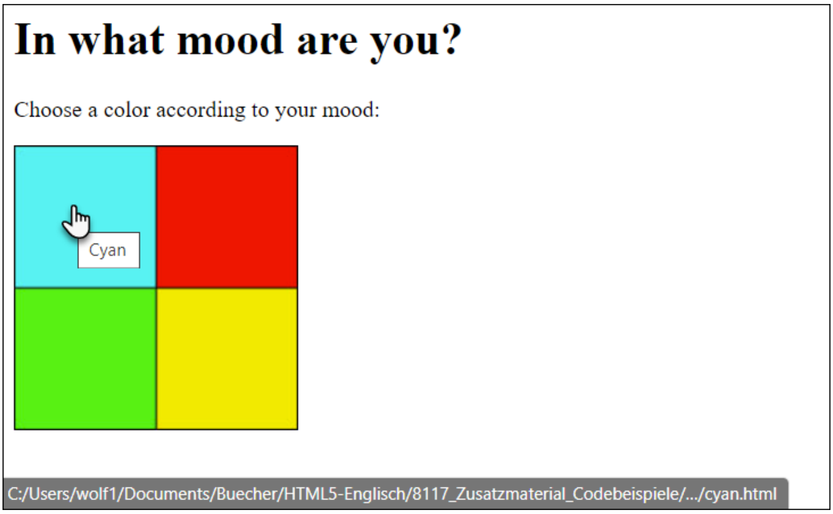
For this purpose, let's first take a look at a simple example in which four rectangular link-sensitive areas with 100 × 100 pixels were defined in a rectangular graphic with 200 × 200 pixels. First, you need to insert the *popart.jpg* graphic into the HTML document. For the anchor name, we use the `#mood` value in the `usemap` attribute. Up to this point, you have inserted just another ordinary graphic. You can introduce the link-sensitive area with the `map` element and the name of the image map (here, `name="mood"`). Between `<map>` and `</map>`, you need to specify the coordinates for the links with the `area` element. You'll find more information about the `area` element after the following example:

```
...
<p>Choose a color according to your mood:</p>
<p>
  <img src="images/popart.jpg" alt="Link-sensitive graphic"
   width="200" height="200" usemap="#mood">
</p>
...
  <map name="mood">
    <area shape="rect" coords="0,0,100,100"
     href="colors/cyan.html" alt="Cyan" title="Cyan">
    <area shape="rect" coords="0,100,100,200"
     href="colors/green.html" alt="Green" title="Green">
    <area shape="rect" coords="100,100,200,200"
     href="colors/yellow.html" alt="Yellow" title="Yellow">
    <area shape="rect" coords="100,0,200,100"
```

```
      href="colors/red.html" alt="Red" title="Red">
  </map>
...
```

Listing 6.4 /examples/chapter006/6_2/index.html

I think the connection between the image where you set the anchor name with the `usemap` attribute and the `map` element that introduces the link-sensitive area should now be clear to you. The `area` element is somewhat more complex. For this reason, the following sections provide a basic description of the `area` element.

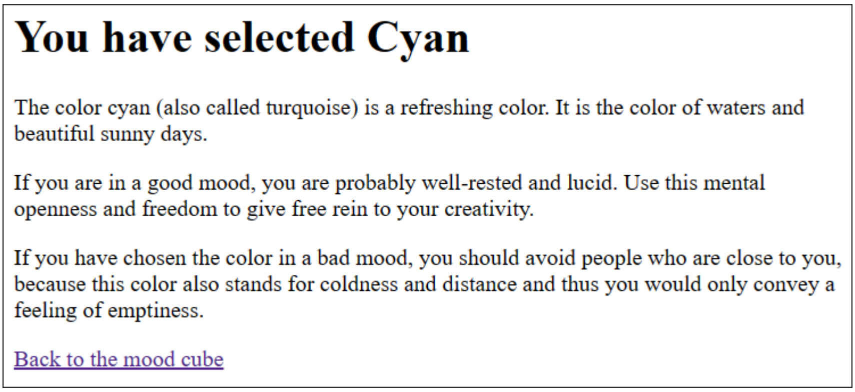


In what mood are you?

Choose a color according to your mood:

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/.../cyan.html

Figure 6.8 Each of the Four Colored Areas Was Linked to a Special Page with Its Own <area> Element; Select the Cyan-Colored Area and the HTML Document cyan.html Will Be Called



You have selected Cyan

The color cyan (also called turquoise) is a refreshing color. It is the color of waters and beautiful sunny days.

If you are in a good mood, you are probably well-rested and lucid. Use this mental openness and freedom to give free rein to your creativity.

If you have chosen the color in a bad mood, you should avoid people who are close to you, because this color also stands for coldness and distance and thus you would only convey a feeling of emptiness.

Back to the mood cube

Figure 6.9 When You Click on the Color, You'll Get Corresponding Feedback on the Selected Color

6.2.1 Use HTML Attributes for the HTML Element <area>

To define the shape of the area type, you need to specify the desired shape in the HTML attribute `shape`. For this purpose, you can use `rect` (= *rectangle*), `circle`, and `poly` (= *polygon*). Besides specifying the shape via `shape`, you also need to specify the coordinates of the link-sensitive area with the HTML attribute `coords` (= *coordinates*). Here, you use absolute values. You can separate multiple values with a comma. With regard to our example, in the following, the values refer to *x1*, *y1*, *x2*, and *y2*.

```
<area shape="rect" coords="100,0,200,100" ...>
```

You can use *x1* and *y1* to define the upper-left corner of the link-sensitive area that extends to the lower-right corner, which you specify with *x2* and *y2*. In this example, we make an indentation by 100 pixels (*x1*) from the top corner of the graphic to the right and 0 pixels (*y1*) down to define the upper-left corner. For the bottom-right corner, we move 200 pixels (*x2*) from the upper-left corner of the image to the right and move 100 pixels (*y2*) down from that position (see Figure 6.10).

If you use `circle` for `shape`, the coordinates are `x`, `y`, `r` (`coords="x,y,r"`). For example, you would need to specify the following coordinates in relation to Figure 6.10:

```
<area shape="circle" coords="100, 100, 50" ...>
```

This way, you move 100 pixels from the upper-left corner to the right (x) and 100 pixels down (y), arriving at the center of the square. From there, you would draw a link-sensitive area with a radius of 50 pixels, which would define a 100-pixel diameter circle from the center as the link-sensitive area.

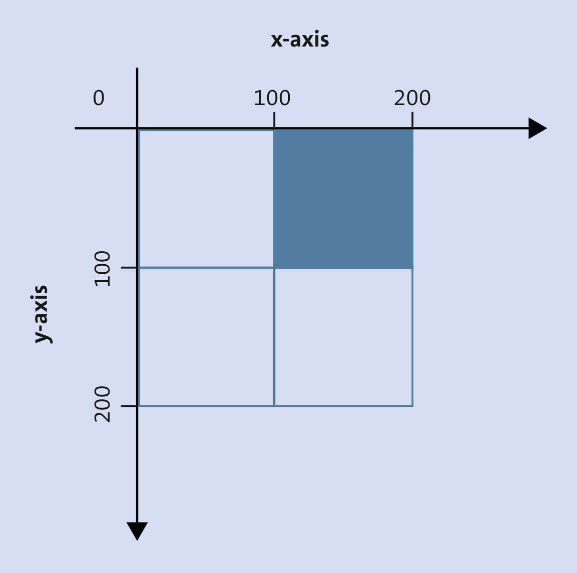


Figure 6.10 The Described Link-Sensitive Area

If, on the other hand, you use `poly` as the `shape`, you can define any number of coordinates in `coords` with *x1*, *y1*, *x2*, *y2* . . . *xn*, *yn*. This enables you to define as many corners as you want. At the end, you have to use the xy coordinates of the first element again to close the polygon.

6.2.2 Referencing Defined Areas of the HTML Element <area>

As with the `a` element, you need to specify the reference target in the `area` element using the `href` attribute. Together with the `href` attribute, you must also write the `alt` attribute in the `area` element.

Pixel Coordinates with a Graphics Program

For more complex link-sensitive areas, such as a map, you can also use special software that allows you to create the link-sensitive area in the corresponding graphic with the mouse and just paste it into your HTML document. If you don't want to install special software for this purpose, you can find a good online image map editor at *www.maschek.hu/imagemap/imgmap*, where all you have to do is upload the image and use your mouse to create the link-sensitive areas.

I added the link-sensitive areas in Figure 6.11 with the nine federal states of Austria using GIMP software and linked them with the official websites, including the website of the capital of the corresponding federal state.

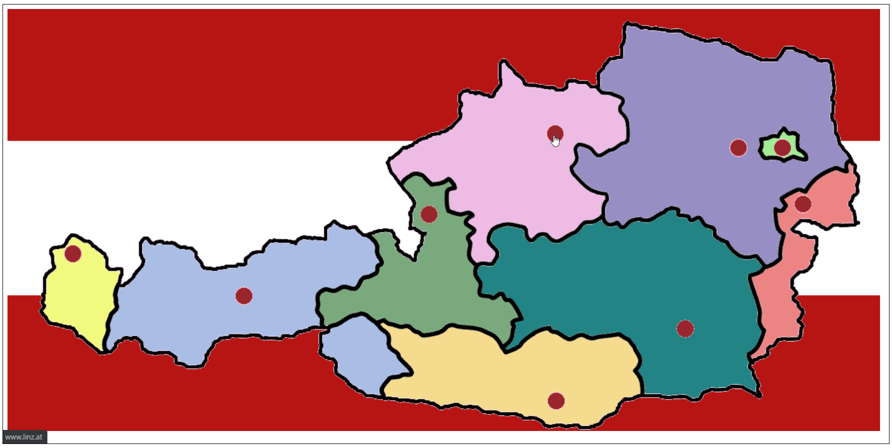


Figure 6.11 Link-Sensitive Areas Can Be More Complex, as in the Case of a Geographical Map

6.2.3 HTML Attributes of <area>

Here's a short overview of the attributes of `<area>`. You'll notice that the `area` element contains the same attributes as a hyperlink with the `a` element.

Attribute	Description
`alt`	You must use the alternative text when you use the `href` attribute.
`coords`	Use this to define the coordinates for the link-sensitive area.
`download`	Use this to provide the referral target in `href` for download.
`href`	Use this attribute to specify the URL of the page referenced by the link-sensitive area.
`hreflang`	Use this to specify the language of the linked document. For this purpose, you can use the typical language abbreviations.
`media`	Use this to specify information about the media for which the link target has been optimized. You can either enumerate media types, separate by commas, or specify `all` for all media types. I'll describe the specification separately in Chapter 7, Section 7.3.7.
`rel`	Return to Chapter 3, Section 3.5, where we already discussed this attribute from the `link` element if you need more information. However, the same applies to the `area` element as to the `a` element, where the `bookmark`, `external`, `nofollow`, and `noreferrer` elements can also be used.
`shape`	Use this to set the shape for the link-sensitive area. Possible values are as follows: ■ `rect`: Rectangle ■ `circle`: Circle ■ `poly`: Polygon
`target`	Use this to specify where the link target should be opened. Possible values for this are as follows: ■ `_blank`: New window/tab ■ `_parent`: Parent window ■ `_self`: Current window ■ `_top`: Top window level ■ `framename`: Name of the window that was opened with JavaScript and also assigned in JavaScript
`type`	Use this to inform the web browser about the multipurpose internet mail extension (MIME) type (file format) to which the linked file belongs. This specification is useful if the target isn't an HTML document.

Table 6.4 The HTML Attributes of the <area> Element

Use Link-Sensitive Graphics?

Since link-sensitive graphics are not responsive, they are rarely used in practice. As an alternative, you can use an SVG graphic where you can specify the appearance of the linked areas with CSS.

6.3 Loading the Appropriate Image Using <picture>

In the past, it wasn't always easy to provide a suitable image for all display sizes. Frequently, quality or performance losses had to be accepted in this regard. The `picture` element enables you to find an HTML element that serves as a container element for multiple image sources. The individual image sources must be specified using the `source` element.

The following example demonstrates the use of these HTML elements:

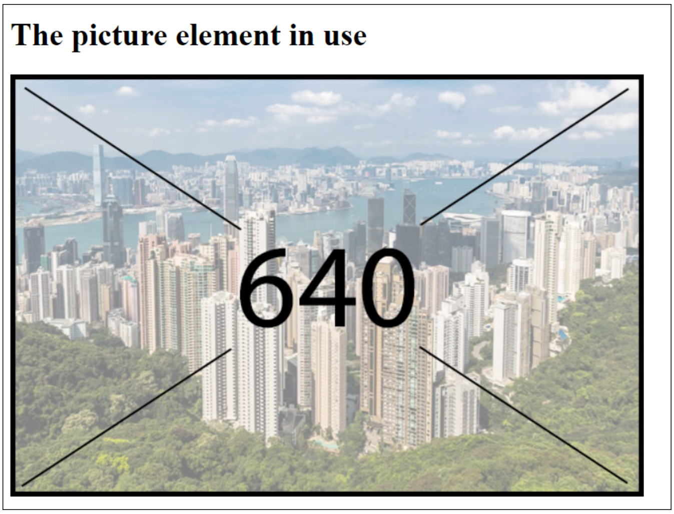
```
...
  <h1>The picture element in use</h1>
  <p>
  <picture>
     <source media="(min-width: 1024px)" srcset="images/HK-1024.jpg">
     <source media="(min-width: 640px)" srcset="images/HK-640.jpg">
     <source media="(min-width: 480px)" srcset="images/HK-480.jpg">
     <!-- Fallback for old browsers -->
     <img src="images/HK-480.jpg" alt="Hong Kong Island">
  </picture>
  </p>
...
```

Listing 6.5 /examples/chapter006/6_3/index.html

Using the `picture` element is relatively simple. Between `<picture>` and `</picture>`, you can position several `source` elements. Each `source` element contains a query (i.e., *media query*) with the HTML attribute `media`, which you can use to query information such as viewport width, viewport height, and orientation. In addition, the `source` element contains the HTML attribute `srcset`, which you use to specify the image file to be loaded. At the end, you should use an `<img>` as a fallback, in case web browsers can't process the `picture` element.

The sources of `<source>` within `<picture>` and `</picture>` will be read from top to bottom, which means that the first match of the HTML attribute `media` will be loaded. If the viewport is at least 1,024 pixels (`min-width: 1024px`), the image *HK-1024.jpg* will get loaded. With a viewport of at least 640 pixels (`min-width: 640px`), *HK-640.jpg* (see Figure 6.12) will get loaded, and with at least 480 pixels (`min-width: 480px`), *HK-480.jpg* will get loaded. Web browsers that can't handle this will load the fallback, that is, `<img>` (here, also *HK-480.jpg*).

These specifications in the HTML attribute `media` are the same as those you use when creating queries in CSS. In addition to `min-width`, you can also use `max-width`, `max-height`, `min-height`, or `orientation` (for alignment). I'll describe such CSS queries, also referred to as *media queries*, separately in Chapter 13, Section 13.4. This section covers only the basic principles of the HTML elements, `<picture>` and `<source>`.



The picture element in use

Figure 6.12 Here, the Screen Width Was at Least 640 Pixels, Which Is Why the Matching Image HK-640.jpg Was Loaded

6.3.1 HTML Attributes of <source>

Here's a listing of the HTML attributes for the `source` element before you'll see more examples about it.

Attribute	Description
`srcset`	You can specify the media source to be loaded. This attribute can also contain multiple media sources. You must separate the individual sources with a comma. In addition to the URL that leads the visitor to the media source itself, you can optionally specify the width with a positive value followed by `w`, and also optionally specify the pixel density followed by an `x` (e.g., `1x`, `2x`, `3x`, etc.).
`media`	You can specify the query (*media query*) for the media sources to be loaded. The attribute can be used only in the `picture` element. If the query specified in `media` is correct, the corresponding source will be loaded into `srcset`. You can learn more about such CSS media queries separately in Chapter 13, Section 13.4.
`type`	You can inform the web browser about the MIME type (file format) to which the linked file belongs. In the example of the `picture` element, for instance, this can be useful if you want to use different file formats for display.
`sizes`	You can specify how wide or with which pixel density the image should be displayed. It's possible to specify additional and more complex framework conditions.

Table 6.5 The HTML Attributes of the <source> Element

6.3.2 Multiple Image Sources with the HTML Attribute “srcset”

You can also specify and reference multiple image sources with the HTML attribute `srcset`. This can be useful if you want to provide an alternative for high-resolution displays besides the ordinary image. With regard to the */examples/chapter006/6_3/index.html* example, you could provide higher resolution images for retina displays with `2x` pixel density as follows:

```
...
  <picture>
     <source srcset="images/HK-1024.jpg, images/HK-1024-hd.jpg 2x"
             media="(min-width: 1024px)">
     <source srcset="images/HK-640.jpg, images/HK-640-hd.jpg 2x"
             media="(min-width: 640px)">
     <source srcset="images/HK-480.jpg, images/HK-480-hd.jpg 2x"
             media="(min-width: 480px)">
     <!-- Fallback for old browsers -->
     <img src="images/HK-480.jpg" alt="Hong Kong Island">
  </picture>
...
```

Listing 6.6 /examples/chapter006/6_3_2/index.html

Using Different File Formats

It’s quite possible to use different file formats using `<picture>`. For this purpose, you should use the `type` attribute in `<source>` to define the MIME type of the image. For example, web browsers such as Chrome or Opera can handle the *WebP* graphics format (see *https://en.wikipedia.org/wiki/WebP*), which is supported as a direct competitor of the JPEG format and requires considerably less memory with comparable quality. Consider this example:

```
...
  <picture>
     <source srcset="images/HK-640-WebP.webp" type="image/webp">
     <source srcset="images/HK-640-jpg.jpg">
     <!-- Fallback for old browsers -->
     <img src="images/HK-640.jpg" alt="Hong Kong Island">
  </picture>
...
```

You can find this example in */examples/chapter006/6_3_2/index2.html*. In Google Chrome and Opera, the WebP graphic will be displayed. All other web browsers will display the JPEG graphic. I haven’t specified the HTML attribute `media` here, although you could use this attribute as well.

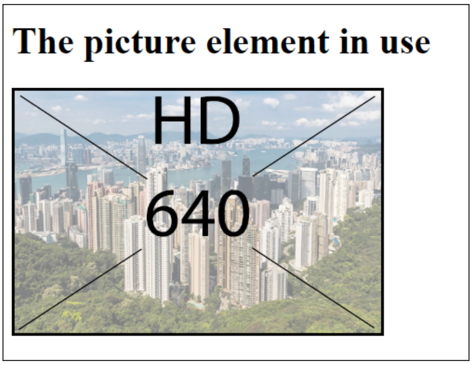

The picture element in use

Figure 6.13 On High-Resolution Displays, the Image Is Loaded with a Higher Pixel Density (Here, “2x”)

6.4 Adding an Icon for the Website (Favicon)

You’ll certainly have noticed the small icons in the address bar, tab, and/or among your favorites quite often. These icons are called *favicons* (short for favorite icons). It’s no big deal to add such an icon to the website. They work well as a (re)identifier for your website. You can specify them in the header of your document using the `link` element, which is described in detail in Chapter 3, Section 3.5.

To do this, you can use the HTML attribute `rel` with the value `icon` and the location and name of the icon with `href`. Optionally, you can use the `sizes` attribute. The size of a favicon is usually 16 × 16 and sometimes 32 × 32 pixels. Moreover, favicons should be saved in *.ico* format. If you use a different data format, you should specify the corresponding MIME type with the `type` attribute (e.g., for PNG format: `type="image/png"`).

Additionally, you can add an icon for mobile devices such as iPhone, iPad, and so on. For this purpose, you need to specify the value `apple-icon-touch` for `rel`, and, correspondingly, the icon should be larger than the favicon. There are various size suggestions for this, such as 76 × 76 for iPhone, 120 × 120 for iPhone with Retina, or 180 × 180 pixels for iPad with Retina. If you use an image that’s too large, it will be scaled accordingly on the device. The icon appears on Apple mobile devices, for example, when you add a web page to the *home screen*. The round corners are added by the operating system itself.

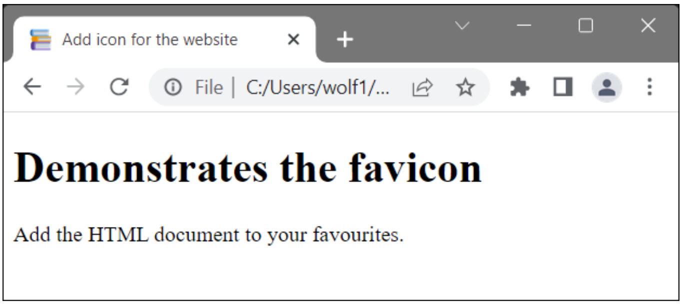
Creating a Favicon

You can create icons or graphics for favicons using any image-editing software. You can also find many online tools on the web that allow you to create and download the icons in the web browser. Websites *http://ionos.de/tools/favicon-generator* and *https://favicon.io* are just two of a multitude of places to start.

Here's an example where a favicon and an Apple icon have been added to the header section of the HTML document using the `link` element:

```
<head>
...
    <link rel="apple-touch-icon" sizes="180x180"
     href="images/apple-touch-icon.png" />
    <link rel="icon" type="image/png" sizes="16x16"
     href="images/favicon-16x16.png" />
</head>
...
```

Listing 6.7 /examples/chapter006/6_4/index.html



Demonstrates the favicon

Add the HTML document to your favourites.

Figure 6.14 Here, You Can See the Favicon in the Top Left of the Table Bar

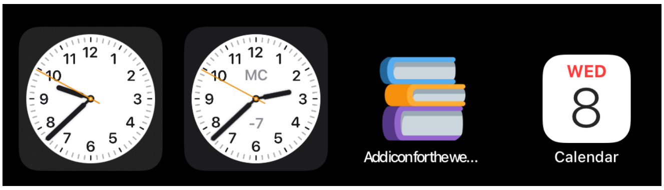


Figure 6.15 The Apple Touch Icon in the Right Side of an iPad

Adding a Favicon to a Website without the <link> Element

Many web browsers are capable of displaying a favicon without the `link` element if you name the files *favicon.ico* and *apple-touch-icon.png* and place them in the web root directory.

6.5 Using Vector Graphics in HTML Documents

If you want to use vector graphics on the web, the XML format called *Scalable Vector Graphics* (SVG) is available for this purpose. The advantage of SVG is that a graphic never loses quality when you zoom in or out. For this reason, an SVG graphic can be printed in high quality on a printer. Unlike other graphics in JPEG or PNG format, for example, you can create and edit SVG files in a simple text editor.

Because SVG is in plain text in an XML syntax and thus platform-independent, you can also create such a graphic in PHP, change it dynamically with JavaScript, or format it with CSS. In practice, this format is very suitable for cartography and illustrations of all kinds, such as logos, icons, scientific charts, program flow charts, technical drawings, and graph visualizations, among others.

6.5.1 Adding SVG as a Graphic Reference Using <img>

The easiest way to include an existing SVG graphic might be to use it as a graphic reference with the `<img>` tag and the `src` attribute. You learned about the `img` element in detail in Section 6.1, and it works with SVG graphics just as it does with ordinary JPEG, PNG, or GIF images, except that you need to include an SVG resource instead.

For this purpose, here's a simple HTML document to which an SVG graphic has been added as a graphic reference with the `<img>` tag:

```
...
<h1>SVG as graphic reference</h1>
<p>
  <img src="logo.svg" width="100" height="50" alt="50%"><br>
  <img src="logo.svg" alt="100%"><br>
  <img src="logo.svg" width="400" height="200" alt="200%">
</p>
...
```

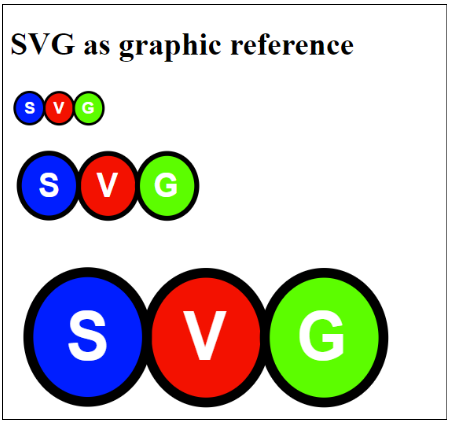
Listing 6.8 /examples/chapter006/6_5_1/index.html

In the example and in Figure 6.16, you can see that the quality of the SVG graphic always remains the same, even if you've scaled the size of the display with the attributes `width` and `height`. Try it out for yourself by setting the width to 4,000 pixels and the height to 2,000 pixels, for example. The original size of the SVG file *logo.svg* is 200 × 100 pixels.

Editors for Editing SVG Graphics

You can create SVG using applications such as Adobe Illustrator (*www.adobe.com/products/illustrator.html*) or the free *Inkscape* (*https://inkscape.org*). There are also tools online, such as *SVG-edit* (*https://github.com/SVG-Edit/svgedit*), which I used to create the SVG graphics for this book. Nevertheless, you can also write and modify SVG completely manually, using an ordinary text editor. After all, SVG is written with XML.

Editors such as Inkscape or SVG-edit are ideal for learning SVG because you can then view the XML code in the text editor and thus better understand the structure of the format.



SVG as graphic reference

Figure 6.16 SVG Added as a Graphic Reference via <img>

6.5.2 Embedding SVG Directly into the Web Page Using <svg>

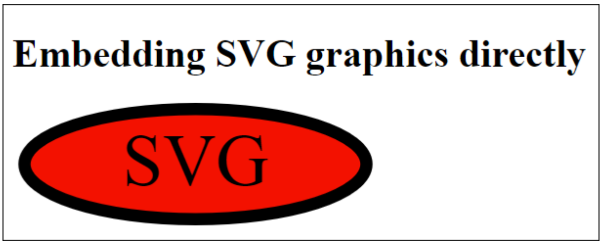
With HTML, it's also possible to embed SVG directly into the HTML document. SVG itself provides different tags, for example, for circles, ellipses, rectangles, polygons, lines, elements for paths, texts, and animations.

You can write the HTML elements `<svg>...</svg>` anywhere in the document body between `<body>` and `</body>`. The SVG syntax with the SVG elements must be written between `<svg>` and `</svg>`. Taking all of this into consideration, a simple example looks as follows:

```
...
<body>
<h1>Embedding SVG graphics directly</h1>
<p>
  <svg width="300" height="100">
    <ellipse ry="45" rx="140" cy="50" cx="150"
             fill="red" stroke-width="10" stroke="black"/>
    <text text-anchor="middle" font-size="60"
          y="67" x="150" fill="black">SVG</text>
  </svg>
</p>
</body>
...
```

Listing 6.9 /examples/chapter006/6_5_2/index.html

In this example, we describe an ellipse filled with red color and black text with the SVG tags `<ellipse ... />` and `<text ... />`. Figure 6.17 shows what it looks like in the web browser.



Embedding SVG graphics directly

Figure 6.17 Here, an SVG Graphic Was Created That's Directly Embedded in the HTML Document

6.5.3 SVG Tags for Vector Graphics

In between `<svg>` and `</svg>`, you must follow the rules for XML documents and keep in mind that the documents are case sensitive. In addition, you must close the standalone SVG tags using the / character, for example, `<circle ... />`). Unlike HTML, SVG syntax is very precise, and an error will usually result in incorrect or even no rendering at all of the SVG graphic.

6.5.4 Overview of Graphical SVG Elements

This section provides a brief overview of the graphical elements in SVG. The goal here is just to show you how to include SVG elements in an HTML document and use them with basic shapes.

Coordinate System

In an SVG graphic, the coordinate system has its origin at the top left. Coordinate data with an x-axis thus points to the right, and those with a y-axis point downward. You define the coordinate system at the beginning in the opening `<svg>` tag via `width` and `height`.

For all graphical SVG elements, you can specify an outline (`stroke-width`; `stroke`; `stroke-linecap`; `stroke-dasharray`), transparency (`opacity`), and fill color (`fill`) using the relevant attributes.

The graphic elements in SVG are as follows:

- `<path .../>`
  A path consists of a line described by various combinations of elliptic arcs, quadratic and cubic curves, and various distances.

- **`<circle .../>`**
  Defines a circle described by a radius (`r`) and the position of the center (`cx`, `cy`). Let's take a look at a simple example:

  ```
  <svg height="200" width="200">
    <circle cx="100" cy="100" r="90"
            stroke="black" stroke-width="3" fill="purple" />
  </svg>
  ```

- **`<ellipse .../>`**
  Defines an ellipse with the two semi-axis radii, which, as with the circle, are described by the position of the center point (`cx`, `cy`). The radius is described with a horizontal (`rx`) and a vertical (`ry`) radius:

  ```
  <svg height="300" width="100">
    <ellipse cx="50" cy="150" rx="45" ry="140"
    style="fill:yellow;stroke:black;stroke-width:3" />
  </svg>
  ```

- **`<rect .../>`**
  The name says it all. This element defines a rectangle over the upper-left corner (`x`, `y`) with `width` and `height`. Rounded corners are also possible. For example, for a simple rectangle you could write the following:

  ```
  <svg width="410" height="200">
    <rect x="5" y="5" width="390" height="150"
          style="fill:red;stroke-width:3;stroke:black" />
  </svg>
  ```

  By the way, the properties within `style` are different from those of classic CSS.

- **`<line .../>`**
  Defines a simple straight line across the coordinates of two endpoints (from `x1`, `y1` to `x2`, `y2`). The following shows a simple example:

  ```
  <svg height="100" width="250">
    <line x1="0" y1="0" x2="250" y2="100"
          style="stroke:blue;stroke-width:3" />
  </svg>
  ```

- **`<polygon .../>`**
  This element allows you to describe a polygon via the different endpoints. You pass the individual x/y points to the `point` attribute. If, on the other hand, you need a line with different points that don't connect at the end, `<polyline .../>` would be the right choice for you. For example, you can implement a simple triangle as follows:

```
<svg width="210" height="200">
  <polygon points="100,5 125,85 80,105"
           style="fill:yellow;stroke:black;stroke-width:5" />
</svg>
```

- **<text ...>...</text>**
  This element allows you to define a text. You specify the position of the beginning of the text with `x` and `y`. Using `font-family`, you can define the font, while `font-size` defines the size of the font. Let's take a look at a simple example:

```
<svg height="100" width="500">
  <text x="5" y="80" fill="red" font-size="60">
    A text with SVG
  </text>
</svg>
```

Units of Measurement and Color Values

By default, the unit of measurement is `px` for pixels if no other specifications have been made. Nevertheless, you can force this unit by writing `pt` (for point), `pc` (for pica), `mm` (for millimeters), `cm` (for centimeters), `in` (for inches), `em` (relative unit of measure), or `ex` (relative unit of measure) after the specified value.

When you specify the fill color via `fill`, the same color values apply as with CSS, and you can use `#FF00FF`, `rgb(100,0,100)`, `rgb(100%, 20%, 80%)`, or just the color name `blue`.

6.5.5 Further Notes on Using SVG

Besides the graphical elements, there's much more you can describe with SVG. For example, there's the option to group multiple graphical elements via `<g>` and `</g>`, which allows you to apply transformations or styling to all elements grouped together.

You can style the elements either via the individual attributes or via the `style` attribute, similar to CSS. A transformation is also available. For example, you can use `transform="rotate(...)"` to rotate text and any shapes. In addition, SVG also provides an animation model.

Furthermore, you'll find filters that allow you to add various effects to the SVG graphic. Thus, blurring, shadows, or illuminations are no problem with SVG. Linear and radial color gradients can also be defined and displayed.

6.5.6 Mathematical Formulas Using MathML

Mathematical formulas can be integrated directly into a web page using HTML via MathML. The implementation of MathML works similar to SVG and is put into effect using existing MathML tags to describe a mathematical formula. The MathML tags need to be written between `<math>` and `</math>`.

Unfortunately, support for MathML hasn't really reached web browsers yet (currently only Firefox and Safari), but using *MathJax* (*www.mathjax.org*), you can fix the problem by writing the following in the HTML document head:

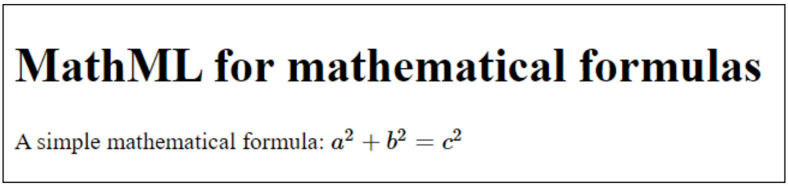
```
...
<script
  src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_
      HTMLorMML">
</script>
...
```

Here's a simple mathematical formula that demonstrates the use of MathML in practice:

```
...
<head>
...
<script
  src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_
      HTMLorMML">
</script>
</head>
...
<h1>MathML for mathematical formulas</h1>
<p>
A simple mathematical formula:
    <math>
      <mrow>
        <msup><mi>a</mi><mn>2</mn></msup>
        <mo>+</mo>
        <msup><mi>b</mi><mn>2</mn></msup>
        <mo>=</mo>
        <msup><mi>c</mi><mn>2</mn></msup>
      </mrow>
    </math>
</p>
...
```

Listing 6.10 /examples/chapter006/6_5_6/index.html

You can see the example in Figure 6.18 during execution.



MathML for mathematical formulas

A simple mathematical formula: $a^2 + b^2 = c^2$

Figure 6.18 The Formula Was Formatted with MathML

I won't go any deeper into MathML and the various tags available because the topic is quite specialized. For a comprehensive overview of MathML, see the W3C website at *www.w3.org/TR/MathML3/*.

6.6 Drawing Graphics Using <canvas>

Originally, the term *canvas* meant work surface. In fact, such an element is initially nothing more than a white area on which you can draw something with the help of JavaScript. Some web developers refer to this element as a programmable `img` element that you can fill yourself pixel by pixel.

Applications and Games with <canvas>

The `canvas` element is being used extensively in the web developer community. There are many interesting examples of diagrams or libraries that include single effects up to entire games. The best way to see for yourself is to visit *www.effectgames.com/demos/canvascycle/* to see what is already made possible by `<canvas>`.

In the introductory `<canvas>` tag, you must use the HTML attribute `id` to specify a document-wide unique name, `width` to specify the width of the canvas, and `height` to specify the height of the canvas. The use of a unique `id` is especially important for accessing the `canvas` element with JavaScript. It's also possible to use multiple `canvas` elements in one HTML document.

For this purpose, here's a simple example in which an empty `canvas` drawing area with 400 × 200 pixels is displayed. For you to see this drawing area in the example, I've also defined a simple frame around the `canvas` element:

```
...
<h1>Creating a drawing area with canvas</h1>
<canvas id="myCanvas" width="400" height="200">
  Your browser does not support the canvas element.
</canvas>
...
```

Listing 6.11 /examples/chapter006/6_6/index.html

With `<canvas>`, you initially provide merely an empty drawing area in the HTML document. In this example, I've additionally inserted an alternative text between `<canvas>` and `</canvas>`, which is only displayed if the web browser doesn't support `<canvas>` or the user has JavaScript disabled. Another option is to use a pixel graphic instead of text, if that makes sense.

Canvas Application Programming Interface

As mentioned at the beginning, the `canvas` element enables you to provide only the drawing area in the HTML document. By means of JavaScript and a corresponding interface (Canvas API), you can fill this area. Here, I've only dealt with the HTML element `<canvas>` as an introduction.

6.7 Playing Videos Using the HTML Element <video>

The `video` element allows you to play a video in the web browser without any special extension. All modern browsers can handle the `video` element.

Currently, the three video formats MP4 (MPEG-4/H.264), WebM (Web Media File), and OGG (Ogg Vorbis) are supported by the `video` element. However, not every web browser can handle all three formats, as there are discrepancies regarding the format licenses of the web browser manufacturers. The most widely used format is MP4, which is also the format most cameras and cell phones use for recording purposes and which every modern web browser can handle. The free formats, OGG and WebM, were intended for use on the web; however, they aren't supported by Safari and Internet Explorer.

If you want to add a video file to the HTML document via the `video` element, you can include it using the `src` attribute as follows:

```
...
<video width="720" src="Dancing.MP4" controls autoplay>
   Your web browser does not support the video tag.
</video>
...
```

Listing 6.12 /examples/chapter006/6_7/index.html

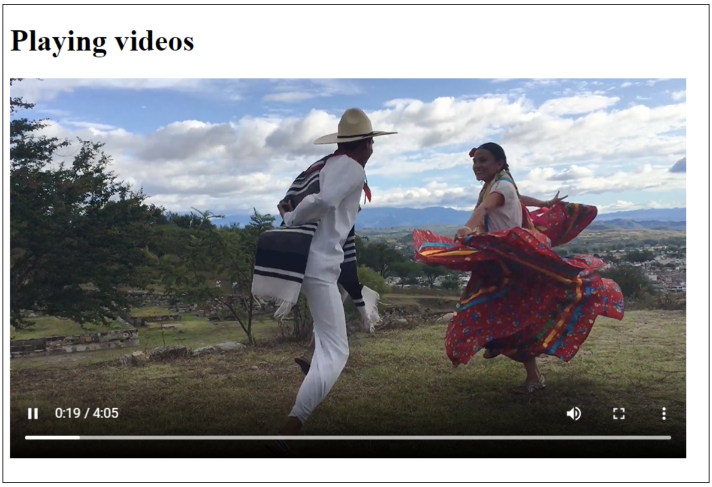
Web Browsers without <video>

Web browsers that don't support the `video` element will display text if you've written one between `<video>` and `</video>`. Alternatively, you could provide a download link in that place.

The */examples/chapter006/6_7/index.html* example is based on the assumption that the web browser supports the MP4 format. If you want to be on the safe side and provide additional formats in case of nonsupport or have a certain format be used preferentially, then you should include additional video files via the `source` element for this purpose. In practice, you can do this as follows:

```
...
<video width="720" controls autoplay>
  <source src="Dancing.webm" type="video/webm">
  <source src="Dancing.MP4" type="video/mp4">
  <source src="Dancing.ogg" type="video/ogg">
  Your web browser does not support the video tag.
</video>
...
```

Listing 6.13 /examples/chapter006/6_7/index2.html



Playing videos

Figure 6.19 Playing a Video Is Hardly a Problem Anymore Thanks to the <video> Element

Inside the `video` element in */examples/chapter006/6_7/index2.html*, you can insert video files via multiple `source` elements using the standalone `<source>` tag. In `<source>`, you can use the `src` attribute to note the video file you want to play. For `src`, the usual rules for referencing in HTML apply. You can use the `type` attribute to indicate the video format. For a list of media types for video formats, see Table 6.6.

Depending on what video format the web browser can handle in the `source` elements, it will play the first known video file. For example, if a web browser knows all the video formats provided, the *Dancing.webm* video will be played in this example. If you want to prefer a video format, you must write it with the `<source>` tag in a higher position between `<video>` and `</video>`.

File Format	Media Type
MP4: MPEG-4 with H264 video codec and AAC audio codec	`video/mp4`
OGG: OGG with Theora video codec and Vorbis audio codec	`video/ogg`
WebM: WebM with VP8 video codec and Vorbis audio codec	`video/webm`

Table 6.6 Video Formats for <video> and the Corresponding Media Types

6.7.1 HTML Attributes for the HTML Element <video>

Most attributes of `<video>` are Boolean attributes and can be used by means of attribute names. In the first example in this chapter, the `controls` and `autoplay` attributes were used, which means that a control (play, pause, volume, etc.) for the video gets displayed, and the video starts automatically after loading. See Table 6.7 for an overview of common HTML attributes you can use for the `video` element.

Attribute	Description
`autoplay`	Starts the video once it has been loaded: `<video autoplay ...>`
`controls`	Displays the controls for controlling the video (play, pause, volume, etc.): `<video controls ...>`
`height`	Sets video height: `<video height="720" ...>`
`loop`	Sets video to play in a continuous loop, restarting as soon as it reaches its end: `<video loop ...>`
`muted`	Sets video to play without sound: `<video muted ...>`
`poster`	Sets a reference to an image that will be displayed until the video gets started: `<video poster="image.jpg" ...>`
`preload`	Specifies how the video should be loaded. With the default setting `preload="auto"`, the entire video is loaded when the page loads. Alternatively, you can use `metadata` to specify that only the metadata gets loaded, whereas `none` makes sure that nothing at all gets loaded along with the loading of the page: `<video preload="none" ...>`
`src`	Specifies the URL of the video file: `<video src="video.mp4" type="video/mp4"...>`

Table 6.7 HTML Attributes for the <video> Element

Attribute	Description
`type`	Allows you to indicate the video format. Possible values can be found in Table 6.6.
`width`	Sets the video width: `<video width="1080" ...>`

Table 6.7 HTML Attributes for the <video> Element (Cont.)

Controlling a Video Using JavaScript

You can also control a video with JavaScript.

6.7.2 Adding Subtitles to a Video Using <track>

Now that HTML makes it possible to render a video consistently, you may want to add more resources to the video. For this purpose, HTML provides `<track>` as an option to define additional tracks. These tracks don't belong to the part of the video or audio track, but simply represent separate text information you can deposit as video subtitles at the exact time of the information contained there for different languages (or for people with a hearing loss/impairment).

By the way, you can use such tracks not only for `<video>` but also for `<audio>`, for instance, to fade in the currently playing soundtrack or the lyrics to a song that's being played. This allows you to provide visitors with hearing loss/impairment with an easier access to the video or audio material. (Note, however, that song lyrics by well-known artists are often subject to copyright.)

The inclusion of additional tracks via the `<track>` tag can be implemented as follows:

```
...
<video width="480" controls>
  <source src="video/Dancing.mp4" type="video/mp4">
  <source src="video/Dancing.ogg" type="video/ogg">
  <source src="video/Dancing.webm" type="video/webm">
  <track default src="subtitles/subtitles_en.vtt" kind="subtitles"
         srclang="en" label="English">
  <track src="subtitles/subtitles_de.vtt" kind="subtitles"
         srclang="de" label="German">
  Your web browser does not support the video tag.
</video>
...
```

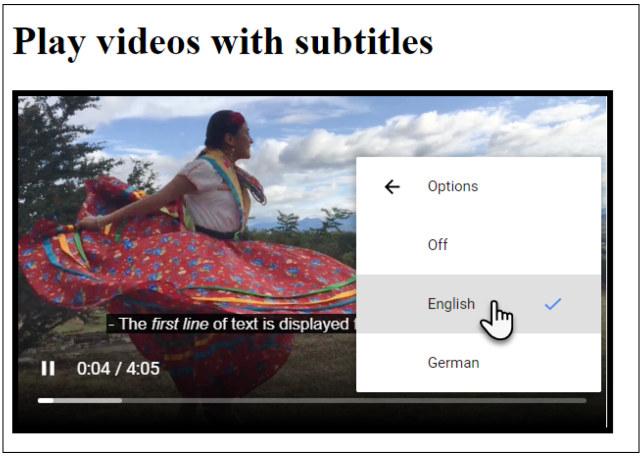
Listing 6.14 /example/chapter006/6_7/index2_subtitles.html

When trying to reproduce the example, you should note that it won't work on a local machine, but must be run online on a web host. You must write the standalone `<track>` tag inside a `video` or `audio` element for which you want to use the additional track. Table 6.8 provides an overview of the individual attributes of `<track>` and their meaning.

Attribute	Description
`default`	Allows you to declare a subtitle to be the default value.
`src`	References the data with the subtitles for the additional track. The data should be in the WebVTT (Web Video Text Tracks) format, which is covered later in this section.
`kind`	Enables you to specify the type of additional track. Possible values are as follows: ■ `subtitles`: Subtitles for spoken or sung content. ■ `captions`: Subtitles of dialogs or sound effects (for people with hearing impairment). ■ `chapters`: Subheadings or subsections that can be jumped to. ■ `metadata`: Descriptions of the data of the audio or video file for scripts or search engines, for example. ■ `description`: Description of the video content (for users with visual impairment).
`srclang`	Sets the language of the text, which should be used for `kind="subtitles"`.
`label`	Sets a title for the text information.

Table 6.8 HTML Attributes for <track>

You can see the example with English subtitles in Figure 6.20 during execution.



Play videos with subtitles

- The *first line* of text is displayed

Figure 6.20 A Video with Subtitles: You Can Choose from the Offered Languages "German" and "English"

Here are a few lines about the WebVTT format, which is used to add additional tracks to a video or audio resource. However, the topic is only briefly touched on here so that you can add your own simple subtitles to a video. For a comprehensive study of the WebVTT format, you should refer to the documentation at *https://www.w3.org/TR/webvtt1/*.

The WebVTT format is saved with the *.vtt* extension. The files can be edited with a simple text editor and saved with UTF-8 character encoding. For instance, the example containing *subtitles.vtt* looks as follows:

```
WEBVTT

1
00:00:01.000 --> 00:00:05.000
The <i>first line</i> of text is displayed for the first 4 seconds.

2
00:00:06.000 --> 00:00:16.000
Now a second line gets
displayed for <b>10 seconds</b>.

3
00:00:17.000 --> 00:00:20.000
This is the final subtitle ...
```

Listing 6.15 /examples/chapter006/6_7/subtitle/subtitles.vtt

The first line contains `WebVTT` followed by a blank line. After that, you can write the content of the file by means of *cues*. Each cue consists of an identification number, a time, a text, and a blank line. The time specification has the following format:

```
hh:mm:ss.mmm --> hh:mm:ss.mmm
```

This specifies that from the time `hh:mm:ss.mmm` to (`-->`) `hh:mm:ss.mmm`, the insertion of the text following it takes place. The time specifications must have a leading 0 if the specification is smaller than 10. `mmm` represents a specification of milliseconds. The other indications are `hh` for hours, `mm` for minutes, and `ss` for seconds. Thus, the meaning of the following lines should be clear:

```
1
00:00:01.000 --> 00:00:05.000
The <i>first line</i> of text is displayed for the first 4 seconds.
...
```

From second 1 to second 5, the text following behind it is displayed, as you can see in Figure 6.20. You can see from `<b>` or `<i>` that you can also design the text visually. Line breaks are also reproduced. There are a lot of other ways to style the text, which I won't go into here. A comprehensive overview of the WebVTT format can be found, as already mentioned, at *https://www.w3.org/TR/webvtt1/*.

6.7.3 Playing Videos via YouTube

Playing videos is no longer a problem thanks to the `<video>` HTML element. But not everyone has the desire, time, or knowledge to convert a video into different formats and deploy it online. In addition, there's still the problem of the increasing volume of data. For example, a video file of 100 MB can cause a lot of traffic if a large number of visitors watch this video on your website. That can quickly add up to a few gigabytes, and not everyone has a fast web host with infinite data volumes.

In this regard, it's a good idea to play the video from YouTube and display it on your website. To do that, you just need to upload the video to YouTube and include the HTML code or link you get from YouTube in the HTML document. Here, you have the choice of using the `iframe` element, the `object` element, or the standalone `embed` element for this purpose. The following example shows all three options in practice, but it only works live on a web server:

```
...
<iframe width="400" height="225"
  src="https://www.youtube-nocookie.com/embed/H8OYVuyBSNA" allowfullscreen>
</iframe>

<object width="400" height="225"
  data="https://www.youtube-nocookie.com/embed/H8OYVuyBSNA" ></object>

<embed width="400" height="225"
  src="https://www.youtube-nocookie.com/embed/H8OYVuyBSNA" >
...
```

Listing 6.16 /examples/chapter006/6_7_3/index.html

You can see the example at execution in Figure 6.21.

Pay Attention to GDPR When Embedding YouTube or Vimeo Videos!

If you embed YouTube or Vimeo videos on your website, you should definitely pay attention to data protection when doing so. When a user starts the video, data is also passed to YouTube. There are a few solutions on how to get around the problem. A great overview of this can be found at *https://www.thomasvantuycom.com/writing/privacy-friendly-video-embeds/*.

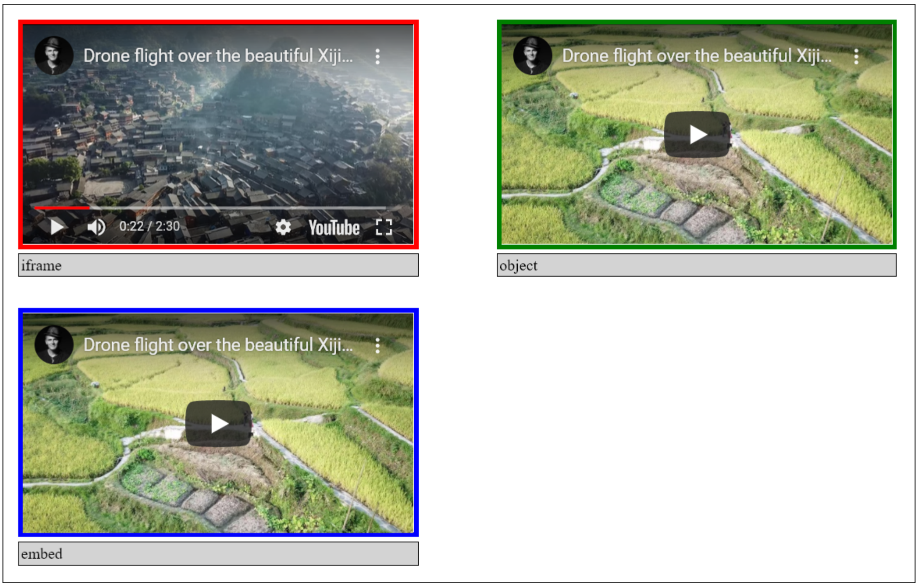

Drone flight over the beautiful Xiji...
0:22 / 2:30
YouTube
iframe
Drone flight over the beautiful Xiji...
object
Drone flight over the beautiful Xiji...
embed

Figure 6.21 Playing a YouTube Video: Examples with <iframe>, with <object>, and with <embed>

6.8 Playing Audio Files Using the HTML Element <audio>

Basically, the HTML element `<audio>` works in the same way as `<video>`, except that you use it to play audio files. Here, too, there are currently three formats: MP3 (MPEG-1/MPEG-2 Audio Layer III), OGG, and WAV (Waveform Audio File), which can be played without additional software using `<audio>`.

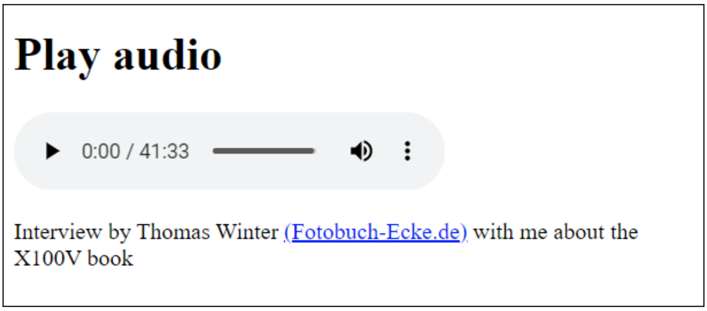
All modern web browsers can handle the MP3 format, and it should probably be the ideal choice apart from the OGG format. The WAV format might be the worst choice for the web most of the time due to the size of the file. If the browser doesn't support the `audio` element, you can write text between `<audio>` and `</audio>` that will be displayed instead. As with `<video>`, you can provide all three formats here via the `source` element.

An example in practice will make clear what I've just described in a little more detail:

```
...
<audio controls>
   <source src="sound/Hello.ogg" type="audio/ogg">
   <source src="sound/Hello.mp3" type="audio/mpeg">
   This web browser does not support the audio tag.
<audio>
...
```

Listing 6.17 /examples/chapter006/6_8/index.html

You can specify multiple `source` elements with the desired audio files between `<audio>` and `</audio>` to offer multiple formats. The web browser uses the first format it supports. Otherwise, the same applies to the `source` element as with the `video` element. For a list of media types (for `type`), see Table 6.9. You can see the example in Figure 6.22 during execution.



Play audio

Interview by Thomas Winter (Fotobuch-Ecke.de) with me about the X100V book

Figure 6.22 Playing an Audio File with the <audio> Element

File Format	Media Type
MP3	`audio/mpeg`
OGG	`audio/ogg`
WAV	`audio/wav`

Table 6.9 Audio Formats for <audio>

6.8.1 HTML Attributes for the HTML Element <audio>

Most attributes of `<audio>` are Boolean attributes and can be used with the attribute name. In the first example in Section 6.8, the `controls` attribute was used, which means that a control (play, pause, volume, etc.) is displayed for the audio to be played. See Table 6.10 for an overview of the common attributes you can use for the `audio` element.

Attribute	Description
`autoplay`	Starts the audio file as soon as it has been loaded: `<audio autoplay ...>`
`controls`	Displays the controls for controlling the audio (play, pause, volume, etc.): `<audio controls ...>`
`loop`	Sets the audio to play in a continuous loop, restarting as soon as it reaches its end: `<audio loop ...>`

Table 6.10 Attributes for the <audio> Element

Attribute	Description
`muted`	Mutes the sound: `<audio muted ...>`
`preload`	Specifies how the audio file should be loaded. With the default setting `preload="auto"`, the entire audio file gets loaded when the page loads. Alternatively, you can use `metadata` to specify that only the metadata gets loaded, whereas `none` makes sure that nothing at all gets loaded along with the loading of the page: `<audio preload="none" ...>`
`type`	Allows you to indicate the audio format. Possible values can be found in Table 6.9.
`src`	Specifies the URL of the audio file: `<audio src="audio.mp3" type="audio/mp3"...>`

Table 6.10 Attributes for the <audio> Element (Cont.)

Controlling Audio Using JavaScript

You can also control audio content with JavaScript.

6.9 Including Other Active Content

Now there's `<video>` for video content, `<audio>` for audio content, and `<img>` for images. But for a number of other types of content, no special HTML element exists, for example, Excel files, AutoCAD, or special movie formats such as QuickTime movies, which aren't provided directly by the web browser, but via an external extension (plug-in) for the web browser. Those extensions aren't integrated in the web browser, and if such an extension is missing, the content can't be executed.

To embed active content in an HTML document that isn't supported by the web browser itself but via extensions, you have at least two helpers available: the `object` element and the `embed` element.

What Can We Use <embed> and <object> For?

Most modern web browsers have removed or disabled direct support for browser plug-ins. It's therefore obvious that it isn't advisable to build a website based on the `embed` or `object` elements if you want the regular visitor to be able to use the website properly as well. Especially things such as Java applets, ActiveX, or Flash, which have become obsolete, should be completely avoided nowadays. For other elements, such as video, audio, and images, the HTML elements `<video>`, `<audio>`, and `<img>` can be used.

6.9.1 HTML Element <embed>

The standalone `embed` element has long been supported by all major web browsers, but even so, it was a relatively late addition to standard HTML. You can use the `embed` element for all possible active elements, which usually requires an extension (browser plug-in). `<embed>` enables you to integrate an object in the HTML document. Because the `embed` element is a standalone element without a closing tag, you can't use alternative text here that gets displayed if the web browser can't handle the content because, for example, no extension is available or installed for it.

Here's a simple example using the `embed` element, where a QuickTime movie has been embedded into the HTML document. However, the movie will only play if the QuickTime plug-in for the corresponding web browser has been installed:

```
...
<h1>Viewing a QuickTime movie</h1>
<embed type="video/quicktime" src="movie.mov" width="640" height="480">
...
```

You can reference the element to embed via the `src` attribute. In practice, you should also specify the height (`height`) and width (`width`) because web browsers can't read this resource themselves. Likewise, it's recommended to specify the MIME type using the `type` attribute, which is `video/quicktime` for a QuickTime movie.

6.9.2 HTML Element <object>

You can also use the `object` element to include active content in the HTML document that isn't supported by the web browser directly, but via extensions. Because `<object>` has a closing tag, it's possible with this HTML element to write alternative content (e.g., text, graphic, or downloadable content) between `<object>` and `</object>`, which will be displayed if the content can't be rendered.

Here's an example of the `object` element:

```
...
    <h1>Playing a QuickTime movie</h1>
    <object width="640" height="480" type="video/quicktime" data="Drone.mov">
      QuickTime can't be played back. Plug-in missing.<br />
      <a href="Drone.mov" download>Download movie</a>
    </object>
...
```

Listing 6.18 /examples/chapter006/6_9_2/index.html

The example is basically equivalent to the example with the `embed` element. However, unlike `<embed>`, `<object>` offers the advantage of being able to provide replacement content or information if the content can't be executed. Nevertheless, it's also better here to avoid browser plug-ins for a general website if possible and to always fall back on the original HTML solutions. This way, you can ensure that any user with any web browser can view or operate the content. Don't make your website dependent on a browser plug-in that users may need to install first or that may not even exist for a specific web browser.

6.9.3 HTML Element <iframe>

The `<iframe>` element represents another possibility to embed something into an HTML document. In practice, this element is mainly used to embed external HTML documents into the current HTML document. While `<iframe>` can be rendered by any web browser, you can still specify alternative text between `<iframe>` and `</iframe>`.

Here's a simple example for the `iframe` element:

```
...
<h1>Using iframe</h1>
<iframe height="320" width="680"
        src="product-placement.html">
  Your web browser does not support iframe!
   <br> Click here for the content:
   <a href="product-placement.html">Product placement</a>
<iframe>
...
```

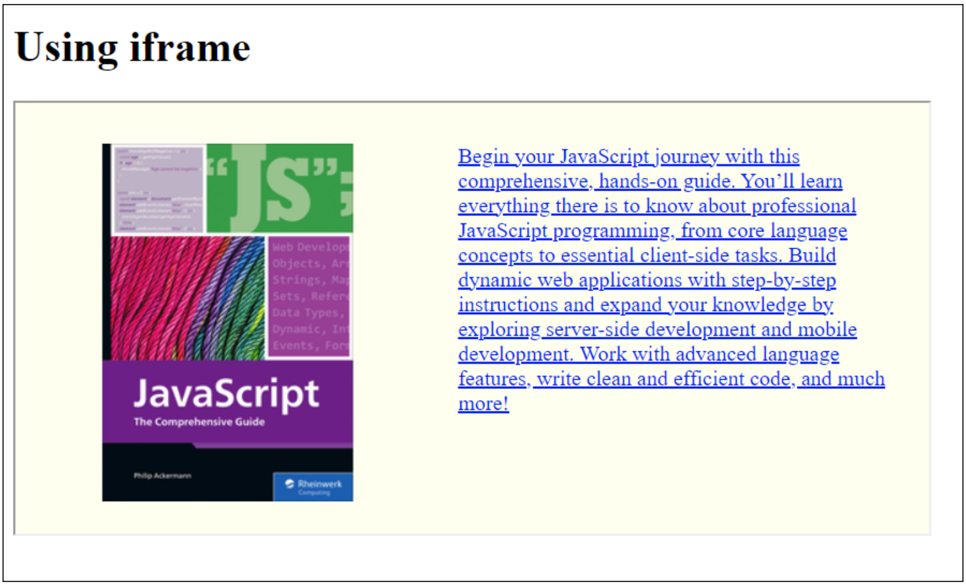
Listing 6.19 examples/chapter006/6_9_3/index.html

Here's the embedded HTML document *product-placement.html*:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Product placement</title>
</head>
<body>
<div>
  <img src="images/cover.png" alt="Cover" style="float:left;">
  <a href=" https://www.sap-press.com/javascript_5554/"
     target="_parent"> Begin your JavaScript journey... </a>
</div>
</body>
</html>
```

Listing 6.20 examples/chapter006/6_9_3/product-placement.html

In this example with `<iframe>`, the HTML file *product-placement.html* is embedded as an advertisement in the current HTML document. You can reference the HTML file via the `src` attribute and specify a size for the frame using the `width` and `height` attributes. You should write the size specification in any case because the web browser can't know how big the embedded HTML document should be. You can see the example in Figure 6.23 during execution.



Using iframe

Begin your JavaScript journey with this comprehensive, hands-on guide. You'll learn everything there is to know about professional JavaScript programming, from core language concepts to essential client-side tasks. Build dynamic web applications with step-by-step instructions and expand your knowledge by exploring server-side development and mobile development. Work with advanced language features, write clean and efficient code, and much more!

Figure 6.23 An HTML Document Has Been Embedded within the Current HTML Document Using <iframe>

<iframe>, <object>, or <embed>

Alternatively, you can achieve the same result by using `<object>` or `<embed>`, and there's basically nothing wrong with that. In practice, however, you should rather prefer `<object>` to `<embed>`. I personally use `<iframe>` to embed HTML documents because it allows me to see more quickly what I want to do here. But that's a matter of personal taste. In addition, `<iframe>` has a few useful attributes (especially `sandbox`) that you don't have when using `<object>`.

When you embed an external HTML document, it may be necessary to restrict certain things for security reasons. Such a restriction is available via the `sandbox` attribute. By using the `sandbox=""` attribute in `<iframe>`, you can prevent scripts from running, links from going out of the frame, plug-ins from being used, cookies from being accessed, and forms from being submitted. You can remove individual restrictions via the following values: `allow-form`, `allow-origin`, `allow-scripts`, and `allow-top-navigation`.

Another standalone attribute of HTML is `seamless`, which not only allows you to embed the specified resource but also to include it in the `src` attribute. The resource included with `seamless` then behaves as if it were a block-generating HTML element in the HTML document, meaning, for example, that the stylesheet definitions also affect the content of the `iframe` resource, which isn't the case if you don't use `seamless`.

6.10 Summary

In this chapter, you learned about some essential HTML elements for adding graphics, videos, and other multimedia content. You are now familiar with the following:

- The `img` element, which you can use to embed images and graphics.
- Link-sensitive graphics (image maps), with which you can embed and use hyperlinks within a graphic.
- How to provide alternative image sources for different viewports using the `<picture>` container element and the `<source>` element it contains.
- How to add vector graphics as a graphic reference or embed them directly into the web page using `<svg>`.
- How to draw something on a web page yourself on the fly using the HTML element `<canvas>` and JavaScript.
- How to play videos with the HTML element `<video>` and audios with the HTML element `<audio>`, as well as which formats are possible in HTML. You also learned how to use JavaScript to control the video or audio. Just in case, you also know how to add a YouTube video to an HTML document.
- The helpers, `<embed>` and `<object>`, with regard to content the web browser isn't able to process and that instead gets executed by extensions (plug-ins). In addition, for embedding other HTML documents into the current HTML document, you have met another alternative to `<embed>` and `<object>`, namely `<iframe>`.

Chapter 7

HTML Forms and Interactive Elements

Nowadays, hardly any major web presence can do without forms (also referred to as HTML forms or web forms), in which visitors can fill out input fields or select something from a list of specific entries, such as a survey form.

Common usage scenarios for HTML forms include contact forms, surveys, signing up for a website or newsletter, guest books, order forms, search functions, adding and uploading data, and more. For such interactions, HTML provides you with various input fields such as text input fields, dropdown lists, radio buttons, and simple buttons.

In this chapter, you'll learn the following:

- How to define and structure forms
- Which input fields are available to you in HTML
- Which HTML attributes exist for input fields and how you can use them meaningfully
- How to send data entered in an HTML form from the web browser to the web server and process it with a PHP script
- Which interactive HTML elements are available

HTML Element	Description
`<form>`	Defines a space for forms
`<fieldset>`	Groups form elements
`<legend>`	Defines a heading for a `fieldset` element
`<label>`	Adds and links a text label to form elements such as input fields, radio buttons, and checkboxes
`<datalist>`	Provides a list of `option` values that can be used as a suggestion list
`<input>`	Requests data from the user via many different types
`<button>`	Defines a clickable button to trigger actions

Table 7.1 Quick Overview of the Elements Covered Here for HTML Forms

HTML Element	Description
`<select>`	Initiates a selection list for which you can create an entry of using the `option` element
`<optgroup>`	Allows you to group `option` elements of a selection list
`<option>`	Defines an entry of a selection or dropdown list
`<textarea>`	Defines an area for entering multiline text
`<output>`	Defines an area for outputting values, and is the counterpart of the `input` element
`<progress>`	Allows you to define an area for the visual representation of a progress
`<meter>`	Visualizes the size of a value

Table 7.1 Quick Overview of the Elements Covered Here for HTML Forms (Cont.)

HTML Element	Description
`<details>` `<summary>`	Expands and collapses page content
`<dialog>`	Displays a dialog box

Table 7.2 Quick Overview of the Interactive Elements Covered Here

7.1 Defining a Space for Forms

You mark an area for an HTML form with the opening `<form>` and closing `</form>`. Everything you write in between the two tags will be part of the form. In practice, various HTML input fields, such as text input fields, selection lists, radio buttons, and labels can be used here.

The basic structure of a typical HTML form could therefore look as follows:

```
<form action="/php/feedback.php" method="post">
    <!--Form elements for the feedback-->
</form>
```

Here you see two typical attributes—`action` and `method`—which are used quite often with HTML forms. The meaning of the individual attributes is as follows:

- `action`
  Specify the URL that will be called upon submitting the form and to which the entered data should be transferred. This is often a PHP script that processes the transmitted data.

- **`method`**
  Specify the *HTTP request method* for how the data should be transmitted to the server for processing. The default setting is `method="get"`, which causes the web browser to append the data as a parameter to the end of the URL. You certainly know this kind of URL, for example, *http://domain.com/search.php?search=HTML*. For larger amounts of data, on the other hand, `method="post"` is more commonly used, which transfers the data to the body of the HTTP request rather than sending it via the URL. I'll cover the topic of processing forms separately in Section 7.6.2 and in Section 7.6.3.

You don't necessarily need to use the two attributes `action` and `method` in the `<form>` tag. The attributes are unnecessary, for example, if you want to process the entered data in the form only with JavaScript, for example, to perform simple calculations. For such purposes, you don't need a URL to be called or an HTTP request method.

Note that the preceding listed attributes weren't all the existing attributes for the `form` element. Other attributes you should know are listed here:

- **`enctype`**
  The default value is the multipurpose internet mail extension (MIME type `"application/x-www-form-urlencoded"`, which means that characters that have a special meaning in a URL are masked with a percentage coding (or URL encoding). For example, a space can be passed as the string `%20`. If you want to upload files, you should use the value `"multipart/form-data"`. Likewise, the value could be just `"text/plain"` for `enctype`, which would allow you to send the form data to an email address right away with `action="mailto:1@woafu.com"`, but that doesn't work very reliably and isn't recommended in practice.
- **`accept-charset`**
  Here you specify which character encoding you want to use to send the data to the web server. With `"utf-8"`, you can enforce UTF-8 as character encoding, for example.
- **`target`**
  Here you can specify the target window in which the web server should output its response. You can use the known values `"_blank"`, `"_self"`, `"_parent"`, `"_top"`, or just the name of a window you created with JavaScript. You already know the meaning of these values from the HTML element `<a>` in Chapter 5, Section 5.2.

7.2 HTML Input Fields for Forms

This section provides an overview of the various classic HTML input fields and the most important attributes for them.

7.2.1 A Single-Line Text Input Field Using <input type="text">

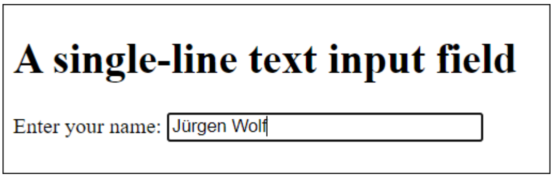
The standalone `input` element allows you to display a single-line text field where users can enter data via the keyboard. Because you can and will use the `input` element for many other field types, you should provide it with the `type="text"` attribute for the single-line text field for clarification (optional). A basic use of `<input>` for a single-line text input field might look as follows:

```
...
<form>
Enter your name:
  <input type="text" name="aName" size="30" maxlength="40">
</form>
...
```

Listing 7.1 /examples/chapter007/7_2_1/index.html

You should use the HTML attribute `name` for an identifier for each input field, because this name is needed by the PHP script when processing the data to access the entered data of the text field. The name is also very useful when you access the form data with JavaScript. You can use the `size` attribute to specify the display length in characters and `maxlength` to specify the actual internally allowed length in characters. You can see the single-line text field in use in Figure 7.1.

If you want to preset the single-line text field with a value and hence visible text, you can do that via the `value` attribute (e.g., `value="John Doe"`).


A single-line text input field

Enter your name: Jürgen Wolf

Figure 7.1 A Single-Line Text Input Field

No Line Break

The `input` element doesn't create a new paragraph, nor does it break the body text, but gets positioned where you write it.

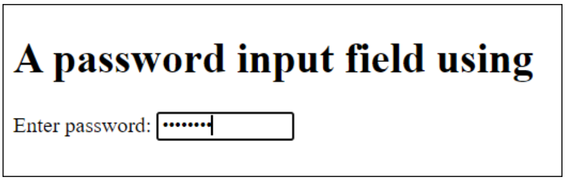
7.2.2 A Password Input Field Using <input type="password">

If you want to create a single-line text field for passwords, you only need to write `type="password"` instead of `type="text"` at the opening `<input>` tag. This will display the entered characters as asterisks or dots. In HTML, you can create a password field as follows:

```
...
<form>
  Enter password:
  <input name="pw" type="password" size="10" maxlength="10">
</form>
...
```

Listing 7.2 /examples/chapter007/7_2_2/index.html

Apart from that, everything that has already been described for the text field applies here as well. You can see the single-line password field in Figure 7.2.



A password input field using

Enter password: ••••••••

Figure 7.2 A Single-Line Text Input Field for Passwords

Of course, you should be aware that a single-line password field doesn't get encrypted and can only serve as a protection against screen readers. The entered password itself is transmitted over the internet via HTTP without encryption in clear text. For encrypted communication between the web browser and the web server, *HTTPS* must be used, and the web server must support this protocol.

7.2.3 A Multiline Text Input Field Using <textarea>

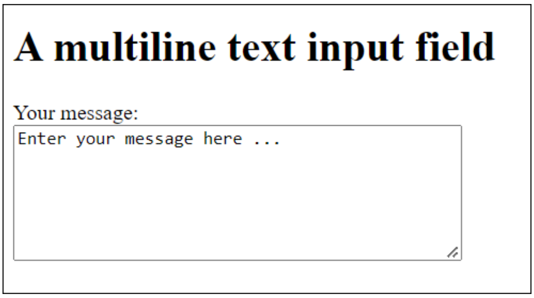
If you need a text field with multiple lines for messages, comments, feedback, or similar, you can use `<textarea>...</textarea>` for that. If you write text between `<textarea>` and `</textarea>`, the multiline text field will be prepopulated with that text. If, on the other hand, you don't want any prepopulated text, you should leave the area in between empty. With HTML, you can write such a multiline text field as follows:

```
...
<form>
Your message:
  <textarea  name="txt" rows="15" cols="50" maxlength="2500">
    Enter your message here ...
  </textarea>
</form>
...
```

Listing 7.3 /examples/chapter007/7_2_3/index.html

In addition, for the multiline text field, you should use the `name` attribute for an identifier because this identifier is needed by the PHP script to access the entered data for processing. The name is also very useful for accessing this data with JavaScript. You can specify the number of displayed rows of the multiline text field using the `rows` attribute and the number of characters per row with the `cols` attribute. It can also be useful to limit the number of characters to be entered with the HTML attribute `maxlength`. You can see such a multiline text input field in Figure 7.3.

By default, the text gets automatically wrapped at the end of the text input field in the next line. The wrap is purely visual and doesn't get sent with the data, unless the user really pressed the Enter key. You can change this behavior via the HTML attribute `wrap`. If you change the default setting of the `wrap="soft"` attribute to `wrap="hard"`, the text will no longer wrap automatically. If the number of characters entered is wider than `cols`, a crossbar will be inserted to scroll sideways. Pressing Enter again will cause a line break.



A multiline text input field

Your message:
Enter your message here ...

Figure 7.3 A Multiline Text Input Field

7.2.4 A Selection List or Dropdown List Using <select>

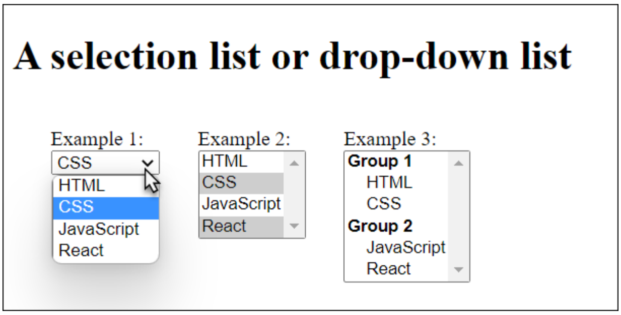
You can formulate a selection list or dropdown list where users can choose from different entries via `<select>...</select>`. The individual options to be selected must be written between `<select>...</select>` with `<option>...</option>`. Here's an example with three different selection lists:

```
...
<form>
  <p>
  Example 1:<br>
  <select name="topic1">
    <option value="val1">HTML</option>
    <option value="val2" selected>CSS</option>
    <option value="val3">JavaScript</option>
    <option value="val4">React</option>
  </select>
  </p>
  <p>
  Example 2:<br>
  <select multiple name="topic2" size="4">
    <option value="val5">HTML</option>
    <option value="val6" selected>CSS</option>
    <option value="val7">JavaScript</option>
    <option value="val8" selected>React</option>
  </select>
  </p>
  <p>
  Example 3:<br>
  <select multiple name="topic3" size="6">
  <optgroup label="Group 1">
    <option value="val9">HTML</option>
    <option value="val10">CSS</option>
  </optgroup>
  <optgroup label="Group 2">
    <option value="val11">JavaScript</option>
    <option value="val12">React</option>
  </optgroup>
  </select>
  </p>
</form>
...
```

Listing 7.4 /examples/chapter007/7_2_4/index.html

You can see all three options in Figure 7.4. In the first example, a classic dropdown list is displayed where one item is visible and can be selected. In the second example, four entries are displayed at once because of `size="4"`. The default setting is `size="1"`, which is used to display a dropdown list like in the first example. The last example also shows how to group entries by writing the corresponding group via `option` elements between `<optgroup label="label">...</optgroup>`. The `label` attribute contains the label for the group.

Again, you should use the `name` attribute in the opening `select` element for the selection lists. If you want to preselect an option, you can do so using the HTML attribute `selected`. If you also want to allow holding down the `Ctrl` or `Shift` key to select multiple options, you need to write the HTML attribute `multiple` in the opening `select` element. If you use a value for `value`, this value will be transferred to the web server in the name-value pair. If, on the other hand, you don't use the `value` attribute in the `option` element, the element content that you've written between `<option>` and `</option>` will be transferred as the value.


A selection list or drop-down list

Example 1:
CSS
HTML
CSS
JavaScript
React
Example 2:
HTML
CSS
JavaScript
React
Example 3:
Group 1
HTML
CSS
Group 2
JavaScript
React

Figure 7.4 Dropdown and Selection Lists in HTML

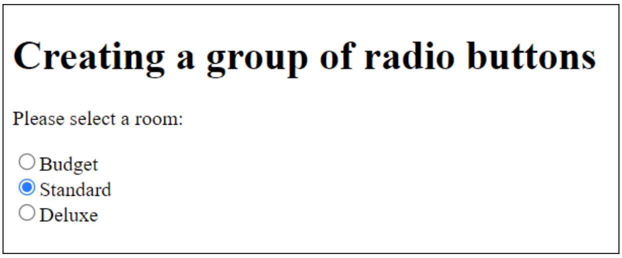
7.2.5 Creating a Group of Radio Buttons Using <input type="radio">

Radio buttons can also be implemented via the `input` element. To do this, you need to specify `type="radio"` in the opening `<input>` tag. Here's a small example of how you can use radio buttons in HTML:

```
...
<form>
  <p>Please select a room:</p>
  <p>
    <input type="radio" name="room" value="budget">Budget<br>
    <input type="radio" name="room" value="standard" checked>Standard<br>
    <input type="radio" name="room" value="deluxe">Deluxe
  </p>
</form>
...
```

Listing 7.5 /examples/chapter007/7_2_5/index.html

You should provide each radio button with the `name` attribute. The radio buttons with the same name belong to a group from which the user can select a value, which is usually the purpose of radio buttons. You can use `value` to specify the value that will be sent to the web server along with the `name` attribute. If you want to preselect a selection, you can do so using the standalone attribute `checked`.


Creating a group of radio buttons

Please select a room:

○ Budget
● Standard
○ Deluxe

Figure 7.5 Radio Buttons in HTML

7.2.6 Adding a Text Label Using <label>

Between `<label>` and `</label>`, you can use a simple text label without any special formatting. That's nothing special at first, but you can use a `label` element to make it easier for the user to operate other elements such as radio buttons or checkboxes. For example, if you've created a radio button as in Figure 7.5 and want to select it, you have to click exactly on the small radio button. You can make this a bit more comfortable and easier for the user by linking the `for` attribute with the `label` element in the case of the radio button. This allows you to select the radio button by clicking on the text label. In practice, you can do this as follows:

```
...
<form>
  <p>Please select a room:</p>
  <p>
    <input type="radio" name="room" id="r1" value="budget">
    <label for="r1">Budget</label><br>
    <input type="radio" name="room" id="r2" value="standard">
    <label for="r2">Standard</label><br>
    <input type="radio" name="room" id="r3" value="deluxe">
    <label for="r3">Deluxe</label>
  </p>
</form>
...
```

Listing 7.6 /examples/chapter007/7_2_6/index.html

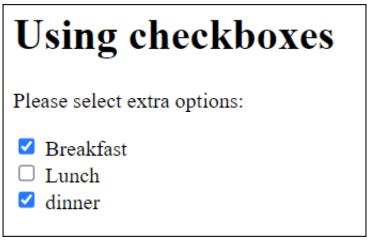
7.2.7 Using Checkboxes via <input type="checkbox">

Checkboxes can also be displayed via the `input` element if you use `type="checkbox"`. Unlike radio buttons, checkboxes allow you to select more than one option. You can create such checkboxes in HTML as follows:

```
...
<form>
  <p>Please select extra options:</p>
  <p>
    <input type="checkbox" name="extra" id="c1" value="breakf">
    <label for="c1">Breakfast</label><br>
    <input type="checkbox" name="extra" id="c2" value="lunch">
    <label for="c2">Lunch</label><br>
    <input type="checkbox" name="extra" id="c3" value="dinner">
    <label for="c3">dinner</label>
  </p>
</form>
...
```

Listing 7.7 /examples/chapter007/7_2_7/index.html

Again, each checkbox should have a `name` attribute with an internal identifier name. You can use `value` to specify the value that will be transferred to the web server when the form is submitted. Figure 7.6 shows the checkboxes in use.



Using checkboxes

Please select extra options:

☑ Breakfast
☐ Lunch
☑ dinner

Figure 7.6 The Checkboxes in Use

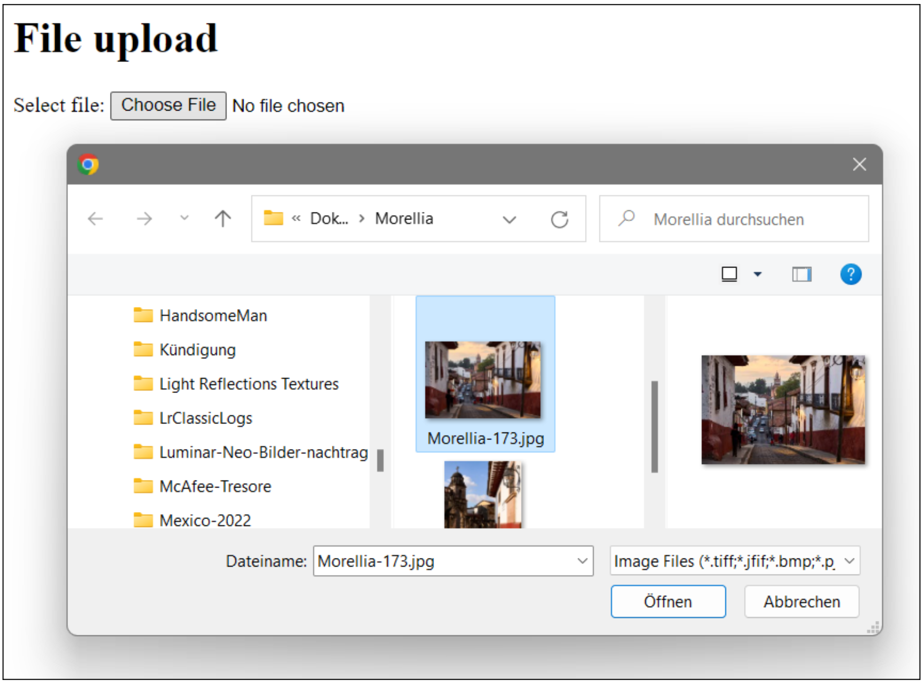
7.2.8 Using Fields for File Uploads via <input type="file">

If you need a field for a file upload, the `input` element is again the first choice. For this purpose, you must use `type="file"`. The web browser usually generates a button that, when clicked, displays the local file selection dialog box. Here's an HTML example of such a file upload:

```
...
<form method="post" enctype="multipart/form-data">
  <p>Select file:
    <input type="file" name="image" accept="image/*">
  </p>
</form>
...
```

Listing 7.8 /examples/chapter007/7_2_8/index.html

If you want to determine not only the file name of the selected file, but the entire file or the contents of the file, you must use `method=”post”` and `enctype=”multipart/form-data”` with the `form` element. Otherwise, you should also use the `name` attribute for an identifier name here. You can also use the `accept` attribute to allow only certain file types to be uploaded. In the example, `image/*` stands for graphics. However, this is merely a filter for the file selection dialog box. In practice, it's still recommended to check the file format on the web server because the `accept` specification isn't strictly followed by every web browser.



File upload

Select file: Choose File No file chosen

Dateiname: Morellia-173.jpg

Image Files (*.tiff;*.jfif;*.bmp;*.p

Dok...
Morellia
Morellia durchsuchen
HandsomeMan
Kündigung
Light Reflections Textures
LrClassicLogs
Luminar-Neo-Bilder-nachtrag
McAfee-Tresore
Mexico-2022
Morellia-173.jpg
Öffnen
Abbrechen

Figure 7.7 The File Upload Dialog Box during Execution

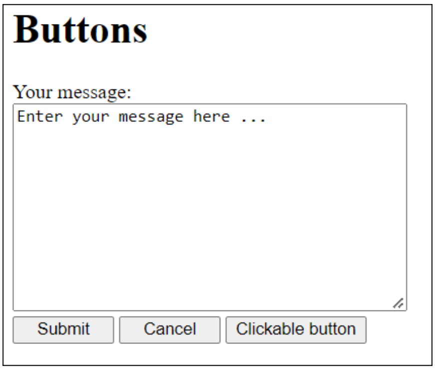
7.2.9 Overview of Various Buttons

Buttons can either be implemented via the `button` element (e.g., `<button>name </button>`), or you can use the two classic options with the `input` element and `type="reset"` or `type="submit"`. The *reset* button (`<input type="reset">`) resets the contents of the form fields within the form to the initial value it contained when the web page was called. The *submit* button (`<input type="submit">`), on the other hand, submits the form and sends the entered data to the URL specified with the `action` attribute in the `form` element. Here's a blank HTML example that demonstrates all three forms of a button:

```
...
<form>
  Your message:<br>
  <textarea  name="txt" rows="15" cols="50" maxlength="2500">
    Enter your message here ...
  </textarea><br>
  <input type="submit" value="Submit" style="width: 80px;">
  <input type="reset" value="Cancel" style="width: 80px;">
  <button type="button">Clickable button</button>
</form>
...
```

Listing 7.9 /examples/chapter007/7_2_9/index.html

You can see the HTML buttons in Figure 7.8.



Buttons

Your message:

Enter your message here ...

Figure 7.8 Buttons in HTML

It's also possible to create the submit or reset button with the `button` element instead of the `input` element by setting the corresponding attributes there with `type="submit"` or `type="reset"`, respectively.

Likewise, you can create a button for scripting with `<input type="button">`. You must specify the name for the button with the `value` attribute, as is the case with the **Submit** and **Reset** buttons. The advantage of `<button type="button">... </button>` over the standalone `<input type="button>` is that you can use other HTML code between `<button>` and `</button>`, which can also be a graphic link.

7.2.10 Using a Hidden Input Field via <input type="hidden">

If you want to include data in a form that isn't visible to users, you can use hidden input fields for this purpose. This is quite useful, for example, to send along additional information or values, or the data calculated or added with JavaScript when sending. For this purpose, too, you can use the `input` element with the `type="hidden"` attribute, for example:

```
<input type="hidden" name="subtotal" value="399">
```

You can specify the data you want to send using the HTML attribute `value`. The identifier for accessing this value of the hidden input field on the web server must be passed along with the `name` attribute.

7.2.11 Writing Form Fields outside of <form>...</form>

You also can write the individual form fields that have been combined into a related form not only inside the related boundary of `<form>` and `</form>` but also outside the related `form` element. It doesn't take much to do that. You only need to use the global `id` attribute in the opening `form` element. HTML input fields for forms that you write outside of the associated `<form>` and `</form>` require the `form` attribute along with the `id` name you wrote in the opening `form` element.

```
...
<form id="form1" method="post" action="/test.php">
    Subject: <input type="text" name="subject"><br>
    Your message:<br>
    <textarea  name="txt" rows="15" cols="50" maxlength="2500">
      Enter your message here ...
    </textarea><br>
</form>
...
<p>
  <input type="submit" value="Submit" style="width: 80px;"
   form="form1">
  <input type="reset" value="Cancel" style="width: 80px;"
   form="form1">
</p>
...
```

Listing 7.10 /examples/chapter007/7_2_11/index.html

7.2.12 Multiple Submit Buttons for Different URLs

It's also possible to set up multiple submit buttons for a form, so that the form can be submitted with different URLs. For these purposes, two new attributes for `input fields` of type `submit` and `image` have been added with `formaction` and `formmethod`, which override the attributes `action` and `method` in the opening `form` element, if those have been written there.

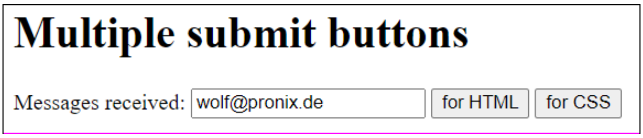
`formaction` allows you to specify the URL that will be called when the submit button is pressed and to which the form data should be submitted. `formmethod` is the HTTP request method that should be used to send the data to the server for processing, that is, either `GET` or `POST`. The meaning of the two attributes corresponds to the `action` and `method` attributes in the opening `form` element, except that each submit button may have its own `formaction` and `formmethod` attribute with different values.

Here's a simple theoretical example:

```
...
<form>
  <label for="email">Messages received: </label>
  <input type="email" name="mail" id="email" required>
  <input type="submit" value="for HTML"
   formaction="/scripts/subscribe-html.php"
   formmethod="post">
  <input type="submit" value="for CSS"
   formaction="/scripts/subscribe-css.php"
   formmethod="post">
</form>
...
```

Listing 7.11 /examples/chapter007/7_2_12/index.html

Instead of `action` and `method` in the opening `form` element, you've written `formaction` and `formmethod` twice each in the `input` field of type `submit`. If you enter an email address here, you can subscribe to either a newsletter for HTML or for CSS news. Either of the two buttons calls a different script.



Multiple submit buttons

Messages received: wolf@pronix.de

Figure 7.9 Two Submit Buttons, Each Calling Different URLs

Other HTML Attributes

In addition to the HTML attributes `formaction` and `formmethod` for the submit button, there are `formenctype` (corresponding to `enctype`) for the encoding type and `formtarget` (corresponding to `target`) for the target window. In addition, the HTML attribute `formnovalid` (corresponding to `novalidate`) can be used as a standalone attribute; this way, you can make sure the input fields won't get validated when submitted.

7.3 Special Types of Input Fields

HTML provides several field types for better input control by the web browser, for example, to specify certain types of data or ranges of values. All new input fields must be written using the `<input>` tag and the corresponding `type` attribute. You already know `<input type="text">` as a single-line text field or `<input type="password">`, which is also a single-line text field where the text is hidden.

Table 7.3 lists the HTML input fields in alphabetical order. An example with all types can be found in */examples/chapter007/7_3/index.html*.

HTML Notation	Description
`<input type="color">`	Displays a control field with a color selection dialog
`<input type="date">`	Displays a control field for a date specification

Table 7.3 HTML Field Types for Controlled Input

HTML Notation	Description
`<input type="datetime">`	Displays a control field for the date, time, and time zone
`<input type="datetime-local">`	Displays a control field for date and time without the time zone
`<input type="email">`	Input field for an email address
`<input type="month">`	Field for entering year and month
`<input type="number">`	Text field for the numbers
`<input type="range">`	Text field for a number within a specific range
`<input type="search">`	Input field for the search
`<input type="tel">`	Input field for phone numbers
`<input type="time">`	Input field for the time
`<input type="url">`	Input field for URLs
`<input type="week">`	Input field for the year and calendar week

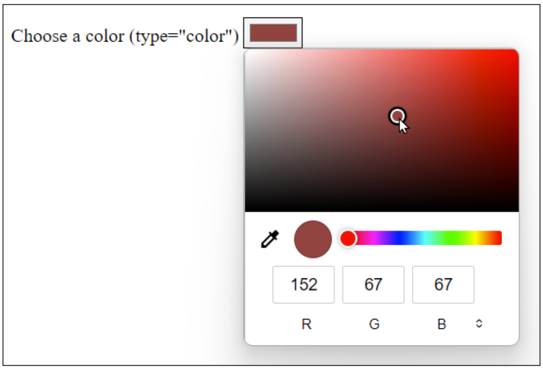
Table 7.3 HTML Field Types for Controlled Input (Cont.)

7.3.1 An Input Field for Colors Using <input type="color">

If you use `<input type="color">`, a user can enter or select a color value via a color selection dialog. To preset the color with a value, you can use the HTML attribute `value` in the following way:

```
<input type="color" value="#FF0000">
```

This way, you've defined an input field for colors, which is preset with red color (hex value: `#FF0000` = red).



Choose a color (type="color")

R 152
G 67
B 67

Figure 7.10 The Input Field for Colors in Windows with the Firefox Browser

The Presentation of the New Input Fields Can Vary

Note that the specification doesn't dictate how web browsers present the various input types. Thus, different web browsers and different systems are likely to mostly use a slightly different user interface of an input field for display.

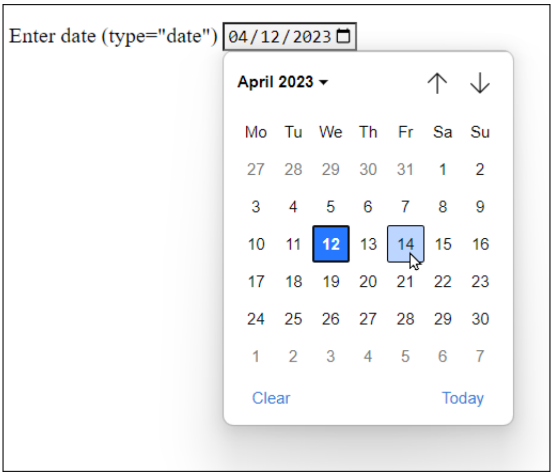
7.3.2 An Input Field for a Date Using <input type="date">

Finally, `<input type="date">` allows you to take users by the hand to query a date. On many websites, you can't tell exactly whether you should enter DD-MM-YYYY, MM-DD-YYYY, or even YYYY-MM-DD. D stands for *day*, M for *month*, and Y for *year*. The new input type `date` opens a selection from which the user can choose the date.

Here, too, you can preset the value with `value` and define the minimum or maximum date specification with `min` or `max`. Write the date in the form of YYYY-MM-DD, for example:

```
<input type="date" value="2023-11-12"
       min="2023-01-01" max="2024-12-30">
```

This sets an input field for a date that defaults to 11/12/2023. In addition, you restrict a valid selection for the date from 1/1/2023 to 12/30/2024. All other entries are invalid.



Enter date (type="date") 04/12/2023

Figure 7.11 The Input Field for a Date in the Chrome Browser

7.3.3 An Input Field for a Time Using <input type="time">

Using `<input type="time">`, you can enter a time in 24-hour format, which will also be validated. An entry such as 27:15 or 22:61 is thus not possible and is recognized as invalid.

If you want to preset the field with a time, you can use the HTML attribute `value`, just as you can use the HTML attributes `min` or `max` to specify an earliest or latest possible time. This specification must be in the format HH:MM, for example:

```
<input type="time" value="15:15" min="08:00" max="17:00">
```

In this example, you've set an input field for a time with 15:15. In addition, you've limited the time with `min` and `max` to 8:00 am and to 5:00 pm, respectively. Everything else is invalid.

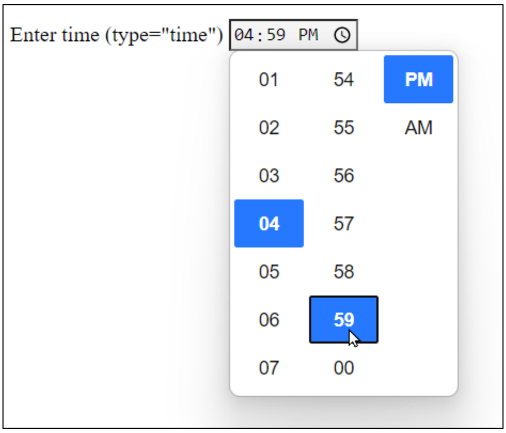


Figure 7.12 Input Field for the Time

7.3.4 Input Fields for Date and Time

You can create a combination of date and time using the input fields `<input type="datetime">` or `<input type="datetime-local">`, where `datetime` is with time zone specification, and `datetime-local` is without time zone specification.

You can preset a value here with the HTML attribute `value` in the format YYYY-MM-DD followed by the capital letter `T` and the time in the format HH:MM, for example:

```
<input type="datetime-local" value="2023-11-12T15:15">
```

This enables you to define an input field for date and time as well as the default with 11/12/2023 at 3:15 pm.

The same is possible with the time zone specification if you set `datetime` as `type`. You must define the time zone at the end with either the capital letter `Z` (for Zulu time) or a specification such as `+0100` (Greenwich +1 hour) or `-0230` (Greenwich –2.30 hours), for example:

```
<input type="datetime" value="2023-11-12T15:15+0100">
```

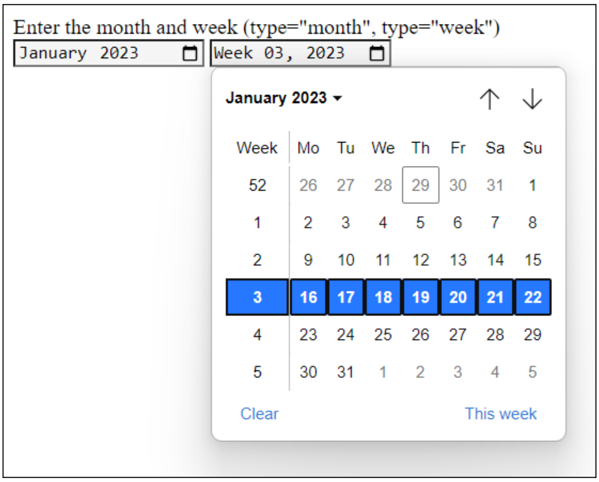
7.3.5 Input Fields for the Month and the Week

Other date-related input fields are `<input type="month">` or `<input type="week">` for the input of a month or a calendar week, respectively, including the year. You can also use the HTML attributes `value`, `min`, and/or `max`. You must specify a month in the form YYYY-MM, for example:

```
<input type="month" value="2023-01">
```

This allows you to define an input field for a month and the year, presetting the input field with January and 2023.

This works similarly for the input field for the week, where you must use YYYY-WW. A year has a maximum of 52 or 53 calendar weeks, and a week begins on Monday.



Enter the month and week (type="month", type="week")

January 2023

Week 03, 2023

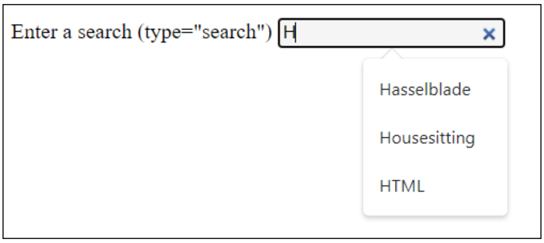
January 2023

Week	Mo	Tu	We	Th	Fr	Sa	Su
52	26	27	28	29	30	31	1
1	2	3	4	5	6	7	8
2	9	10	11	12	13	14	15
3	16	17	18	19	20	21	22
4	23	24	25	26	27	28	29
5	30	31	1	2	3	4	5

Figure 7.13 Input Fields for the Month and the Week in Use

7.3.6 An Input Field for Searches Using <input type="search">

You can define an input field for a search term using `<input type="search">`. Visually, such fields are rendered for searches such as ordinary text fields (`type="text"`).



Enter a search (type="search") H

Hasselblade

Housesitting

HTML

Figure 7.14 Search Input Field

Only when the user starts typing something into the text field does a small “x” appear on the right-hand side of the input field, which enables the user to quickly delete the search term. However, the exact display here also depends on the implementation in various web browsers. In any case, you have a semantic solution for a search input field: `<input type="search">`.

7.3.7 An Input Field for Email Addresses Using <input type="email">

You can define an input field for email addresses via `<input type="email">`. When you do this, the web browser will check if the email address has a valid format. For example, you can style this field with the CSS pseudo-classes `:valid` or `:invalid` to check whether the entered email address is a valid one, for example:

```
...
<style>
    input:invalid { background: red; }
    input:valid { background: ivory; }
<style>
...
<input type="email" multiple requried>
```

In this example, the input field gets displayed in red color (`:invalid`) if the field doesn’t contain a valid email address. If the entered address is valid, the input field will be displayed in ivory (`:valid`).

The HTML attribute `required` is also used very often with this input field because, without this option, an empty field could be submitted to the web server. The HTML attribute `multiple` also makes it possible to specify more than one email address in the field. The individual email addresses are separated by commas.

Pseudo-Classes “:valid” and “:invalid”

The two pseudo-classes aren’t limited to `type="email"`, but can also be used with other input types. However, I’ve jumped the gun here anyway with CSS regarding a topic I won’t go into more detail until Chapter 8.

7.3.8 An Input Field for a URL Using <input type="url">

You can use `<input type="url">` to define an input field for web addresses. As with `type="email"`, the web browser performs a simplified validation of the URL. Other web browsers, on the other hand, list the most recently visited web pages from the history here.

7.3.9 An Input Field for Phone Numbers Using <input type="tel">

`<input type="tel">` allows you to define an input field for a phone number. However, no special format is recommended here, nor is there any validation. You aren't even restricted to enter only digits, and you can also use typical additional characters as in +1-801-123-4567. Smartphones that know this input field display the keyboard for a phone number input at this point. If you really need a specific validation, you must use the `pattern` attribute to implement it. A great place to start with many ready-made `pattern` attributes can be found at *http://html5pattern.com*.

7.3.10 An Input Field for Numbers Using <input type="number">

For a manual input of numbers, you can use the input type `<input type="number">`. Again, the web browser validates the input to see if it's a number or not. And again, you can style this field using the CSS pseudo-classes `:valid` or`:invalid`, for example, to check whether or not it's a valid number. Many web browsers display this input field with a *spinbox* and also the HTML attributes `min`, `max`, and `step` are commonly used. The `step` attribute enables you to specify by how much the value will increase when you use the rotation field.

Both positive and negative numbers are allowed for input. This also applies to floating point numbers that must be used with a period as the decimal point. The exponential notation with e+, E+, e-, or E- is also allowed.

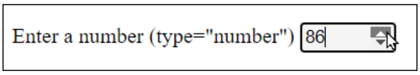


Figure 7.15 An Input Field for Numbers

7.3.11 An Input Field for Numbers of a Certain Range

To implement a slider that can also be used for entering numbers, you can use `<input type="range">`. Usually, the HTML attributes `min`, `max`, and `step` are used to allow a value in a certain range.

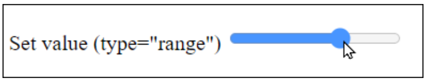


Figure 7.16 A Slider for Entering Numbers

7.3.12 Outputting Values and Calculations Using <output>

The HTML element `<output>` can be used to output values or the result of a calculation and is something like the counterpart of `<input>`. Here, we'll demonstrate the element together with the slider that has been defined using the `input` element (`type="range"`):

```
...
  Adjust slider:
  <input type="range" id="ival" value="50">
  <output id="oval" for="ival">50</output>
...
```

To demonstrate the `output` element in practice at all, I had to anticipate a few things here. Without JavaScript, updating the `output` element is impossible. For this purpose, I added the following JavaScript:

```
…
   function synchOutput(ev) {
          document.getElementById('oval')
              .value = ev.target.value;
      }
      document.getElementById('ival')
          .addEventListener('input', synchOutput);
…
```

Via `addEventListener()`, we virtually listen at the `input` element with the ID of `ival` and call the `synchOutput()` function when a change occurs, as we set the value of the `output` element with the ID of `oval` according to the slider. However, this description is strongly abbreviated. JavaScript will be discussed separately later on in this book.

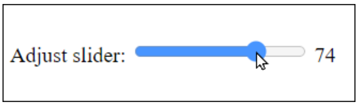


Figure 7.17 The <output> Element Outputs the Current Value of the Slider

7.4 The HTML Attributes for Input Fields

You'll also find HTML attributes for the input fields, which helps you to avoid JavaScript validations and to give the user a helping hand with the input. A first brief overview of the attributes can be found in Table 7.4.

HTML Attribute	Description
`autofocus`	The field should receive the focus when loading.
`autocomplete`	This (de)activates autocompletion of a field or a complete form.
`list`	This allows a list of predefined values to be used for `input` fields.

Table 7.4 Attributes for Input Fields

HTML Attribute	Description
`max` `min`	This allows you to set a maximum or minimum value for the input field.
`multiple`	Multiple values can be specified in one field.
`pattern`	This checks an input against a regular expression passed to `pattern`.
`placeholder`	A text is displayed as a placeholder until the user clicks on the field.
`required`	For this attribute, the input field must be filled out so that it can get passed on by the web browser.
`step`	This controls the step level for some input fields.

Table 7.4 Attributes for Input Fields (Cont.)

7.4.1 Setting the Input Focus Using the HTML Attribute "autofocus"

If you pass the `autofocus` attribute to an input field, the field will immediately receive focus when loaded, for example:

```
...
<form>
  Text 1 <input type="text" autofocus><br>
  Text 2 <input type="text"><br>
  Text 3 <input type="text">
</form>
...
```

Listing 7.12 /examples/chapter007/7_4_1/index.html

When this form is loaded in the web browser, the focus is immediately in the first input field, and the user can start typing immediately. However, you should only provide one input field on a web page with this attribute.

7.4.2 (De)activating Autocompletion Using the "autocomplete" Attribute

Almost all current web browsers use some kind of autocompletion for form data. You can use the `autocomplete` attribute to control whether user input can be saved during the completion process. By default, most web browsers have this service enabled (`autocomplete="on"`). With `autocomplete="off"`, you can prevent saving the input. By the way, passwords aren't stored. Here you can either define the attribute in `<form>`, and the elements it contains inherit the `autocomplete` status, or you can use the `autocomplete` status for individual `input` elements as well.

7.4.3 A List of Suggestions for Using the HTML Attribute “list” and <datalist>

The `list` attribute for `input` fields allows you to suggest a list of possible values for the input. You can define such a list in turn using the `datalist` element. The `datalist` element gets the `list` value you set in the `input` field as `id`. Let’s take a look at a simple example:

```
...
<form>
  Title <input list="mylist" name="title"><br>
    <datalist id="mylist">
      <option value="Mr.">
      <option value="Mrs.">
      <option value="Professor">
    </datalist>
</form>
...
```

Listing 7.13 /examples/chapter007/7_4_3/index.html

Depending on the web browser, you’ll either find a small dropdown menu on the right-hand side of the input field when the field where the suggestions are listed gets the focus. Or a pattern matching the list could be suggested below it during input if `autocomplete` hasn’t been disabled.

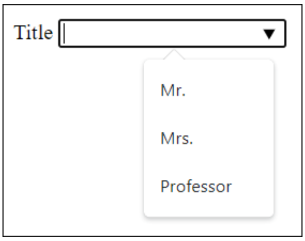


Figure 7.18 A List of Suggestions for the <input> Field

7.4.4 Specifying Minimum and Maximum Values and the Step Size

I’ve already described the HTML attributes `min`, `max`, and `step` several times, for example, when entering numbers with `type="number"` or the slider `type="range"`, and I’d like to mention them again briefly here. I’ve also shown you, when specifying date and time, that you can use `min` and `max` for other values beyond ordinary numbers.

With `min` or `max`, you can define the permitted value range for the input type. Thus, the form will never submit a value that’s less than `min` and greater than `max`. The `step` attribute in turn is used to control the step level of the input, for example:

```
<input type="number" value="50" min="0" max="100" step="5">
```

Only a value between `0` and `100` can be entered in this input field. The rotating field is incremented or decremented by the value `5`.

7.4.5 Selecting or Entering Multiple Values Using “multiple”

The Boolean attribute `multiple` can be used with `<input type="file" multiple>` and `<input type="email" multiple>`, allowing the user to upload multiple files or enter multiple email addresses.

7.4.6 Regular Expressions for Input Fields Using “pattern”

If you want to restrict input using regular expressions, you can do so via the `pattern` attribute. A classic example is the entry of a five-digit number, as is required in some countries for postal codes. You can formulate such an input field with `pattern` as follows:

```
<input type="text" pattern="[0-9]{5}">
```

The input field is only filled in correctly and transmitted if the pattern that gets entered consists of five digits.

Some input types, such as `email`, `url`, or `number`, already have something like a built-in regular expression built; for them, the web browser checks if the entered format matches the input type. For more commonly used patterns for the HTML attribute `pattern`, visit *http://html5pattern.com*.

7.4.7 A Placeholder for an Input Field Using “placeholder”

Another help for the user is placeholder text, which enables you to specify a hint in an input field that disappears when the input field gets the focus, for example:

```
...
User <input type="text" placeholder="Username"><br>
Password <input type="password" placeholder="Password"><br>
...
```

Listing 7.14 /examples/chapter007/7_4_7/index.html

Many web browsers set the specified text as a gray placeholder, as shown in Figure 7.19, for example.

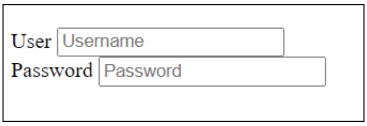


Figure 7.19 The Placeholder in Use

7.4.8 Defining an Input as Required Using the “required” Attribute

The Boolean attribute `required` is used when an input field must be completed. As long as no input has been made, the web browser won’t submit the form.

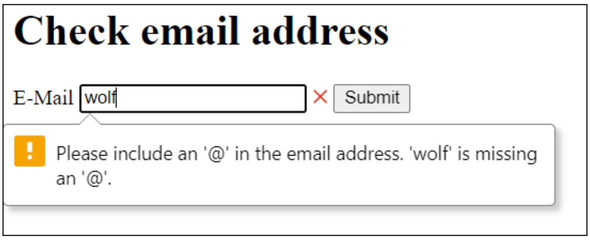
7.4.9 Controlling Error Messages for Input Fields

Some HTML elements check for the validity of the input. For example, if you enter an invalid email address for the `email` input type, the data won’t be sent to the server. Let’s look at a simple example:

```
...
<form>
  Email <input type="email" placeholder="Email" id="em">
  <label for="em"></label>
  <input type="submit">
</form>
...
```

Listing 7.15 /examples/chapter007/7_4_9/index.html

If, in this example, you enter an invalid email address and submit the data using the **Submit** button, you may receive the error message shown in Figure 7.20. However, this message depends on the web browser and operating system and is difficult to change.



Please include an '@' in the email address. 'wolf' is missing an '@'.

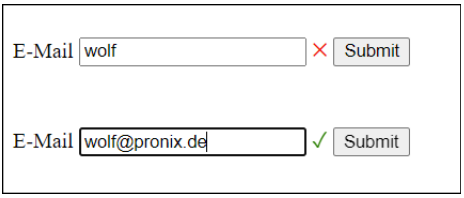
Figure 7.20 The Input Was Invalid

As I just demonstrated with the input type `email`, other input types such as `number`, `url`, or `pattern` work as well. When the user enters something into such input fields, the web browser recognizes whether the input is valid or invalid. You can format this status using the CSS pseudo-classes `:valid` and `:invalid`. In terms of the email address, you could write the following with CSS:

```
...
<style>

  input[type='email']:invalid + label::after{
    color:red;
    content: " ×";
  }
  input[type='email']:valid + label::after{
    color: green;
    content: " ?";
  }

<style>
...
```

Listing 7.16 /examples/chapter007/7_4_9/index.html

In this very simple example, a small red “x” will display at the end of the input field during input if the value of the entered email address is still invalid. If the email address is valid, a green check mark will display at the end of the input field. You could have used `input:invalid` and `input:valid` here without the input type `email`, but then all other input fields (if any) would be considered as well. In the example, an empty `label` element was used to indicate behind the input field with a green check mark or the red “x” whether the input is correct or incorrect.

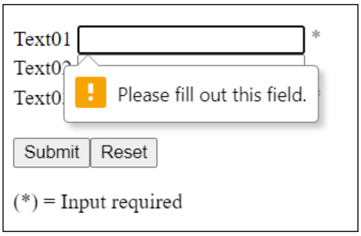


E-Mail wolf ✕

E-Mail wolf@pronix.de ✓

Figure 7.21 Invalid and Valid Email Addresses

The same happens if you’ve provided a field with the attribute `required`. Again, the error message depends on the operating system and web browser if you submit a form where an input field with `required` hasn’t been filled out.



Text01 *

Text02

Text03

Please fill out this field.

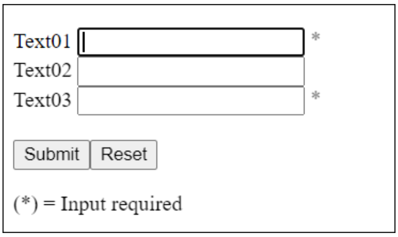
(*) = Input required

Figure 7.22 The Input Field Was Provided with the “required” Attribute

Again, you can use the CSS pseudo-class `:required` to style these input fields separately with CSS. A simple example would be to put an asterisk after each of these input fields with the indication that an input in this field is required in any case, for example:

```
...
<style>
  input:required + label::after{ color: gray; content: " *"; }
<style>
...
<form>
  Text01 <input type="text"  id="t1" required>
  <label for="t1"></label><br>
  Text02 <input type="email" id="t2">
  <label for="t2"></label><br>
  Text03 <input type="email" id="t3" required>
  <label for="t3"></label><br><br>
  <input type="submit"><input type="reset">
</form>
<p>(*) = Input required
...
```

Listing 7.17 /examples/chapter007/7_4_9/index2.html



(*) = Input required

Figure 7.23 An Asterisk Indicates Which Fields Require Input

Deactivating the Validation

It's also possible to use the Boolean attribute `novalidate` to avoid a validation for individual input fields.

7.5 Other Useful Helpers for Input Fields

This section introduces a few more useful HTML attributes and elements for the input fields. For an initial overview, see Table 7.5.

Attribute	Description
`disabled`	Disables input elements

Table 7.5 Useful HTML Attributes and an HTML Element for Input Fields

Attribute	Description
`readonly`	Sets the input element as read only, not editable
`tabindex`	Presets a tab order when the Tab key gets pressed
`accesskey`	Determines a shortcut for a form element
`<fieldset>` `</fieldset>`	Groups multiple form elements

Table 7.5 Useful HTML Attributes and an HTML Element for Input Fields (Cont.)

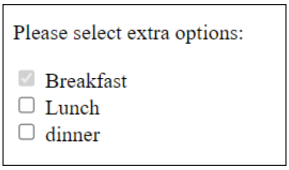
7.5.1 Disabling Form Elements Using the HTML Attribute "disabled"

You can completely disable a form element using the Boolean HTML attribute `disabled`. This element is usually displayed in gray or in a paler font to make it clear that it can't be clicked or edited. This attribute can be used in all form elements such as input fields, dropdown lists, buttons, and radio buttons. Let's take a look at a simple example:

```
...
<p>Please select extra options:</p>
<p>
  <input type="checkbox" name="extra" id="c1" value="breakf"
         checked disabled>
  <label for="c1">Breakfast</label><br>
  <input type="checkbox" name="extra" id="c2" value="lunch">
  <label for="c2">Lunch</label><br>
  <input type="checkbox" name="extra" id="c3" value="dinner">
  <label for="c3">dinner</label>
</p>
...
```

Listing 7.18 /examples/chapter007/7_5_1/index.html

In this example, the checkbox for breakfast has been deactivated. This option isn't available as an extra, but should still be visible and can't be deselected. The other two checkboxes, on the other hand, are optional.



Please select extra options:

☑ Breakfast
☐ Lunch
☐ dinner

Figure 7.24 The Checkbox for "Breakfast" Has Been Deactivated

7.5.2 Permitting Read-Only for Input Fields Using the “readonly” Attribute

The `readonly` attribute is also a Boolean standalone attribute and can be used to mark an input field as read only. This attribute is useful for input fields where users should see the content but can’t change it (e.g., taxes, nationality, result of a calculation). This way, you virtually turn an input field into an output field. Although users can no longer edit the field, they can still select its contents and copy it to the clipboard, for example.

7.5.3 Useful Keyboard Shortcuts and Tab Sequence for Input Fields

You can also guide the user by means of the attributes `tabindex` and `accesskey`. With the HTML attribute `tabindex`, you can use the `Tab` key in the HTML form to jump to the individual form elements in the order you specified via `tabindex`. The form element with the lowest value is jumped to first, followed by the second lowest, and so on to the form element with the highest `tabindex` value.

The HTML attribute `accesskey` allows you to specify a keyboard shortcut the visitor can press to jump to a form element. However, the keyboard shortcuts are usually different in each web browser. For example, if you assign `accesskey="a"` to a form element, you can jump to it by pressing `Alt` + `A` in some web browsers. Other web browsers, in turn, require `Ctrl` + `Alt` + `A`, `Ctrl` + `A`, or `Alt` + `Shift` + `A`.

Let’s take a look at a simple example:

```
...
<form>
  <label>Text01</label>
  <input type="text"  id="t1" placeholder="Name"
   tabindex="2" accesskey="n"><br>
  <label>Text02</label>
  <input type="email" id="t2" placeholder="Email"
   tabindex="1" accesskey="e"><br>
  <label>Text03</label>
  <input type="text" id="t3" placeholder="Country"
   tabindex="3" accesskey="c"><br><br>
  <input type="submit" tabindex="4">
  <input type="reset" tabindex="5">
</form>
...
```

Listing 7.19 /examples/chapter007/7_5_3/index.html

Here, the `Tab` key would first jump to the second input field because it has the lowest `tabindex` value. Pressing the `Tab` key again would lead to the first input field with the `tabindex` value `2`. By pressing the `Tab` key again, you would continue with the third input field, and so on.

You can also access the individual input fields via keyboard shortcuts. With N, you control the first; with E, the second; and with C, the third input field. All of this is to be regarded in relation to the corresponding key combination of your web browser, such as Ctrl + Alt + N in Google Chrome.

7.5.4 Grouping Form Elements Using <fieldset> and <legend>

It can also be very useful to group multiple form elements between `<fieldset>` and `</fieldset>`. In between, you can combine multiple form elements into one visual group. Many web browsers frame this area with a line, for example, to make this group visually clear. Nevertheless, the use of `<fieldset>` isn't intended for visual formatting, which, as we know, you should never do with HTML elements, but purely for logical grouping. For visual formatting, you should always use CSS. You can also set a heading for this group of elements by using `<legend> ... </legend>`. Let's take a look at a simple example:

```
...
<form>
  <fieldset>
    <legend><h2>Your data</h2></legend>
    <label>Name</label>
    <input type="text"  id="t1" placeholder="Name"><br>
    <label>Email</label>
    <input type="email" id="t2" placeholder="Email"><br>
    <label>Date of birth</label>
    <input type="date" id="t3">
  </fieldset>

  <fieldset>
    <legend><h2>Input</h2></legend>
    <input type="submit">
    <input type="reset">
  </fieldset>
</form>
...
```

Listing 7.20 /examples/chapter007/7_5_4/index.html

In Figure 7.25, you can see the grouping in the web browser.

Heading in <legend>

Previously, plain text had to be used within a `legend` element. Since the release of HTML 5.2, heading elements (`h1`, `h2`, `h3`, etc.) can also be placed here. This is useful when you use a group of different sections in a form, which in turn is extremely useful for users who depend on the document outline for navigation.

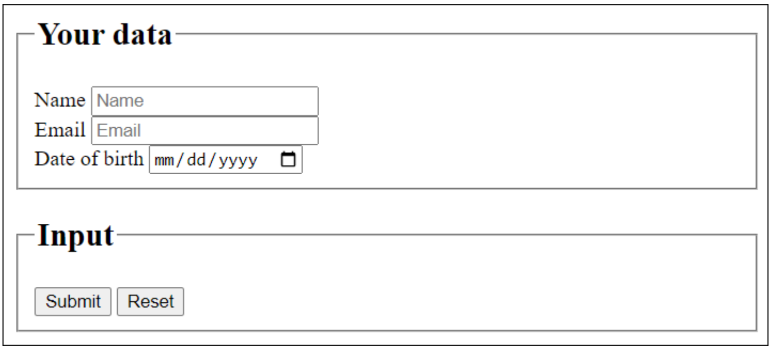


Figure 7.25 A Form with Grouped Form Elements

7.5.5 Progress Display via <progress>

You can use the `progress` element to define a progress indicator to show the progress of an action such as downloading a file or filling out an HTML form. Figure 7.26 shows such an indicator, which may look different depending on the browser used.

```
...
<p>
  Progress bar:
  <progress value="33" max="100">Progress: 33%</progress>
</p>
...
```

Listing 7.21 /examples/chapter007/7_5_5/index.html

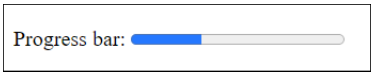


Figure 7.26 Progress Display via <progress>

In practice, you can use the `progress` element along with JavaScript to visualize the state of a work via the progress bar accordingly. To account for accessibility, the current state of the progress bar should be written between `<progress>` and `</progress>`.

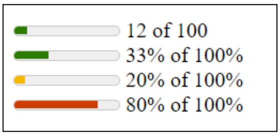
The HTML attribute `value` enables you to specify the number of processed steps, while `max` indicates how many steps are possible at maximum. The progress bar will be displayed according to these two values. The value of `value` mustn't be greater than that of `max`.

7.5.6 Visualizing Values Using <meter>

Another HTML element you can use to represent values visually is the `meter` element. The element can be used, for example, to display various measured values. Figure 7.27 shows the `meter` element in use, which in turn can look different depending on the web browser used.

```
...
<p>
  <meter value="12" max="100">12 of 100</meter> 12 of 100<br>
  <meter value="0.33">33% of 100%</meter> 33% of 100%<br>
  <meter value="10" min="0" low="25" high="75" max="100">
    20% of 100%
  </meter> 20% of 100%<br>
  <meter value="80" min="0" low="50" optimum="25" high="75" max="100">
    80% of 100%
  </meter> 80% of 100%<br>
</p>
...
```

Listing 7.22 /examples/chapter007/7_5_6/index.html



12 of 100
33% of 100%
20% of 100%
80% of 100%

Figure 7.27 Display of Measured Values with <meter>

The `meter` element is often used together with JavaScript to adjust and visualize a measured value accordingly. Again, for the sake of accessibility, you should write the current measurement value between `<meter>` and `</meter>`.

The `value` attribute enables you to define the current measured value, while `max` indicates which maximum value is possible. The measured value is visualized according to these two values. The value of `value` mustn't be greater than that of `max`. If no `max` value is used, this attribute contains the default value `1`. Via `low`, you can set an upper limit of the lower measuring range. The counterpart, which sets the lower limit of the upper measuring range, is `high`. You can specify an optimum value with `optimum`; it must be between `min` and `max`.

7.6 Sending Form Data Using PHP

Using a real example, I'll now demonstrate how data from an HTML form entered in a web browser gets sent to the web server and is processed there. To do this, we'll use a simple form mailer in PHP, without me going into more detail about PHP here. I've deliberately kept this section very simple, as I don't presume that you have PHP knowledge. My only concern is that you understand how the data from an HTML form gets transferred to the web server for further processing.

Styling Forms Using CSS

I've already styled the form we'll be using in the following example via CSS. To learn how to style forms yourself using CSS, see Chapter 14, Section 14.7.

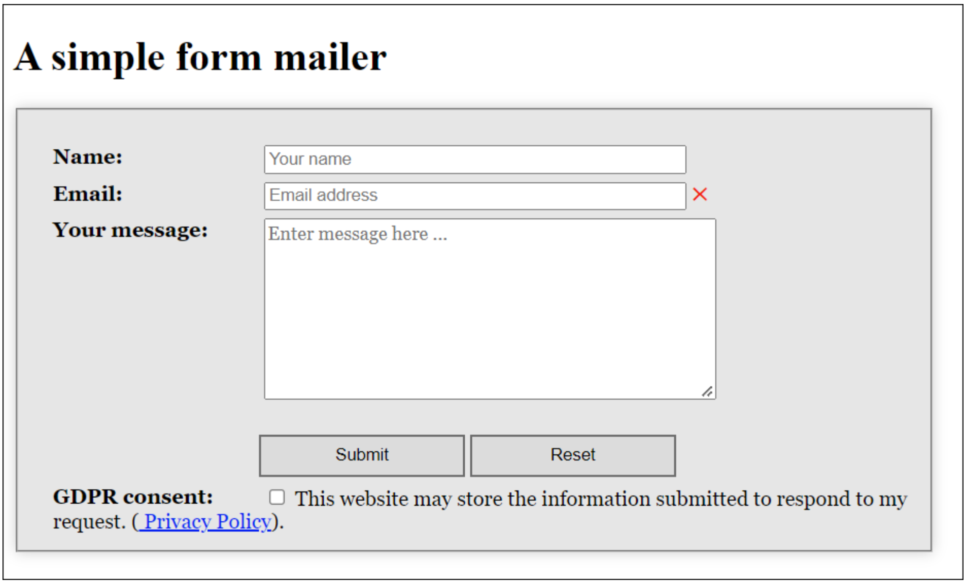
7.6.1 Transferring the Data from the Web Browser for Further Processing

When a user fills in the input fields of an HTML form and clicks the **Submit** button to send the data, the data is transferred to an application on the web server for further processing. You've specified which application that is via the `action` attribute in the opening `<form>` tag. In practice, that's often a script (usually in PHP), which is able to process the form data transmitted by the web browser. To avoid becoming too theoretical, let's create a simple form mailer for this purpose:

```
...
<form id="myForm" method="post" action="form-mail.php">
  <fieldset>

    <div>
      <label for="name">Name:</label>
      <input type="text" name="name" id="name"
             placeholder="Your name" required>
      <label id="error1"></label>
    </div>
    <div>
      <label for="mail">Email:</label>
      <input type="email" name="mail" id="mail"
             placeholder="Email address" required>
      <label for="mail" id="error2"></label>
    </div>
    <div>
      <label for="nachricht">Your message:</label>
      <textarea name="message" id="message"
                placeholder="Enter message here ..."
                rows="8" required></textarea>
      <label for="message" id="error3"></label>
    </div>
    <div>
      <input name="submit" type="submit" value="Submit">
      <input name="Reset" type="reset" value="Reset">
    </div>
   <div>
     <label for="gdpr" id="error4">GDPR consent:</label>
     <input type="checkbox" id="gdpr" name="gdpr" required />
     <label>This website may store the information submitted to respond to
            my request. (<a href="#" target="_blank">
            Privacy Policy</a>).
     </label>
    </div>
  </fieldset>
</form>
...
```

Listing 7.23 /examples/chapter007/7_6/index.html

The data in this HTML form will be submitted to a PHP script named *form-mail.php*, which in this example is located in the same directory as the HTML file containing the HTML form on the web server.



A simple form mailer

Name: Your name

Email: Email address ✕

Your message: Enter message here ...

GDPR consent: ☐ This website may store the information submitted to respond to my request. (Privacy Policy).

Figure 7.28 A Simple HTML Form Mail

You can use the `action` attribute in the opening `<form>` tag to specify where the form's data should go. The web browser compiles an *HTTP request* (a request variant) from the input in the form. Such a request usually consists in a simplified way of the method name, the path to the requested resource, and the HTTP version, for example:

```
GET http://address.com/script.php HTTP/1.1
```

In addition to the script to be called, you can also specify an HTTP request method, if you send form elements via HTTP. The method you use to submit this string to the web server is specified by the `method` attribute in the opening `<form>` tag. If you don't specify a method, the default setting, that is, the `GET` method (`method="get"`), will be used. There are several HTTP request methods, the most important of which are `GET` and `POST`.

7.6.2 The "POST" Method

The `POST` method (`method="post"`) is primarily used for larger amounts of text. Here, when content is requested, the data is transferred from the form in a data block of name-value pairs. Unlike the `GET` method, this HTTP request method sends the message in the separate body of the HTTP request and is therefore not visible in the URL. In our form mailer, the `POST` method was used for the `method` attribute. An advantage of `POST` is that the length of the data is usually unlimited. And if you plan to upload files, this is also possible only with the `POST` method.

7.6.3 The "GET" Method

With this HTTP request method, the web browser appends the form data to the address specified via the HTML attribute `action` using a `?` character at the end as a *query string*. With regard to our example, the URL looks as follows after sending with the `GET` method:

```
http://www.internetaddress.com/form-mail.php?name=Jason+Wolf&mail=wolf%40
pronix.com&message=Hello+Jason%21%0D%0A%0D%0AGreat+website%21%0D%0A%0D%0A
Greetings%0D%0AJohn&send=send&gdpr=on
```

What looks quite chaotic here at first has a structure after all. The question mark separates the URL from the query string that contains the data. An `=` in the query string separates the name from the values, and an `&` separates the individual name-value pairs. If you split the URL, including the query string, at these points, the whole thing looks a bit clearer:

```
http://www.internetaddress.com/form-mail.php
name=Jason+Wolf
mail=wolf%40pronix.com
message=Hello+Jason%21%0D%0A%0D%0AGreat+website%21%0D%0A%0D%0ARegards%0D%0AJohn
send=send
gdpr=on
```

All other cryptic characters are just encodings. For example, a space is replaced by `+` or `%20`. The @ sign is displayed as `%40`, and so on. If you also decode these characters, you'll get a full overview of the name-value pairs that are transmitted with the form to the web server as an HTTP request:

```
name=Jason Wolf
mail=wolf@pronix.com
message=Hello Jason!

Great website

Regards John
send=send
gdpr=on
```

The `GET` method is more commonly used for small amounts of data, such as a search query. If the amount of data is more extensive or you don't want the `GET` parameters to be displayed in readable form in the URL, you can use the `POST` method. In addition, the length of the data is limited with `GET`. However, you can bookmark the event page of a `GET` form because all the necessary information is available in the URL and query string.

GDPR Consent

If you use a form mailer, then you should also add the GDPR consent as a checkbox, as it was done in the example. The European Union stipulates that users must actively agree to the storage of the transmitted data. A link to the privacy policy should also be added.

7.6.4 Processing the Data Using a PHP Script

In the example, the form data is transferred to the PHP script, *form-mail.php*. Even though this book doesn't cover programming with PHP scripting, the corresponding listing is shown here with a brief explanation. If you want to try this example on a web server or web host, you need to make sure that the script is executable for users, which is why you may need to adjust the execution permissions for the *form-mail.php* file. Likewise, PHP must be usable on your web host.

```
<?php
// Data for the configuration
$mailto      = 'youraddress@address.com';
$mailFrom    = 'form-mailer PHP script';
$mailSubject = 'Feedback from PHP form';
$returnPage  = 'http://serveraddress/thankyou.html';
$returnError = 'http://serveraddress/error.html';
$mailContent = '';

// Read form data and create mail from it
if(isset($_POST)) {
  foreach($_POST as $name => $value) {
   $mailContent .= $name . " : " . $value . "\n";
  }
}
// Send email
$mailSent = mail( $mailto, $mailSubject, $mailContent,
                  "From: " . $mailFrom );
// Check email dispatch
if($mailSent === TRUE) {
  header("Location: " . $returnPage);
}
else {
  header("Location: " . $returnError);
}
exit();
?>
```

Listing 7.24 /examples/chapter007/7_6/form-mail.php

I intentionally kept the script very short. Initially, you should adjust the data for the configuration. At the very least, you should assign your email address to the `$mailTo` variable so that you can receive the submitted form data. In the `foreach` loop, the script's task is to read the submitted `POST` data with the name-value pairs (here, with `$name` and `$value`) and create content for the email with `$mailContent`. This email is sent using the `mail()` function to the address in the `$mailTo` variable, with the subject in `$mailSubject` and the contents of the `$mailContent` variable with `$mailFrom` as the sender.

Next, you want to check the return value of the `mail()` function you saved in `$mailSent`. The value can be either `TRUE` (in case of success) or `FALSE` (in case of an error). Depending on whether or not the form was successfully sent, a corresponding redirection to the address you specified in `$returnPage` or `$returnError` takes place. It's recommended to specify a full URL. For this purpose, the *thankyou.html* file was added, which gets displayed if the form could be sent successfully:

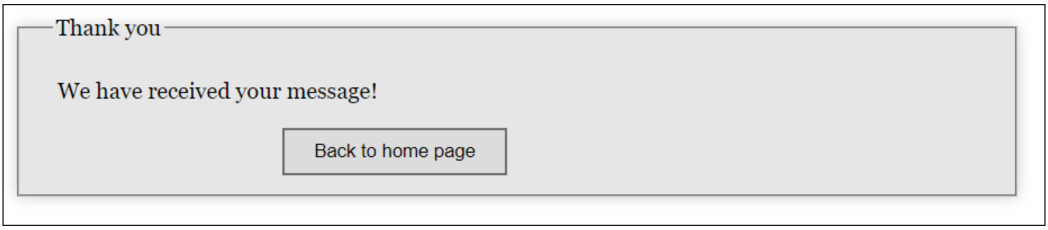
```
...
<form id="thankyou" action="http://serveraddress/">
  <fieldset>
  <legend>Thank you</legend>
    <div>
      <label>We have received your message!</label><br><br>
      <input type="submit" value="Back to home page">
    </div>
  </fieldset>
</form>
...
```

Listing 7.25 /examples/chapter007/7_6/thanks.html

The *error.html* file gets displayed if an error occurs after submitting the form.

```
...
<form id="thankyou" action="index.html">
  <fieldset>
    <legend>Error during data transmission</legend>.
    <div>
      <label>The data could not
             be sent to us!</label><br><br>
      <input type="submit" value="Back to the form">
    </div>
  </fieldset>
</form>
...
```

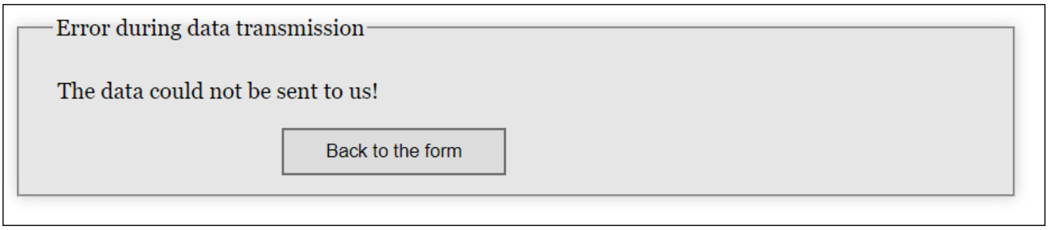
Listing 7.26 /examples/chapter007/7_6/errors.html



Thank you

We have received your message!

Figure 7.29 The Form Has Been Successfully Submitted



Error during data transmission

The data could not be sent to us!

Figure 7.30 An Error Occurred after Submitting the Form

This very simple example can also be used for other forms and is suitable for any number of input fields. However, I haven't considered input fields here where multiple selections can be made at the same time (e.g., `<select multiple>`). In addition, this script isn't suitable for uploading files because the submitted files would have to be checked.

When using transmitted and security-critical data, you should also include a check. For example, it would have been possible to use `$_POST['mail']` to pass the sender to `$mailFrom` right away without any problem, but there are certainly users who could use this line to manipulate the email header.

7.7 Interactive HTML Elements

HTML also provides elements for interactive content. Interactive elements can be changed and adjusted by visitors or can be interacted with in other ways.

7.7.1 Expanding/Collapsing Content Using <details> and <summary>

The `details` element allows you to expand and collapse page content. This can be useful if there's too much detail and information at once, and you don't want to overwhelm visitors with it. They can then expand and collapse the additional information by clicking on a `summary` element. Until now, such a task had to be implemented with JavaScript.

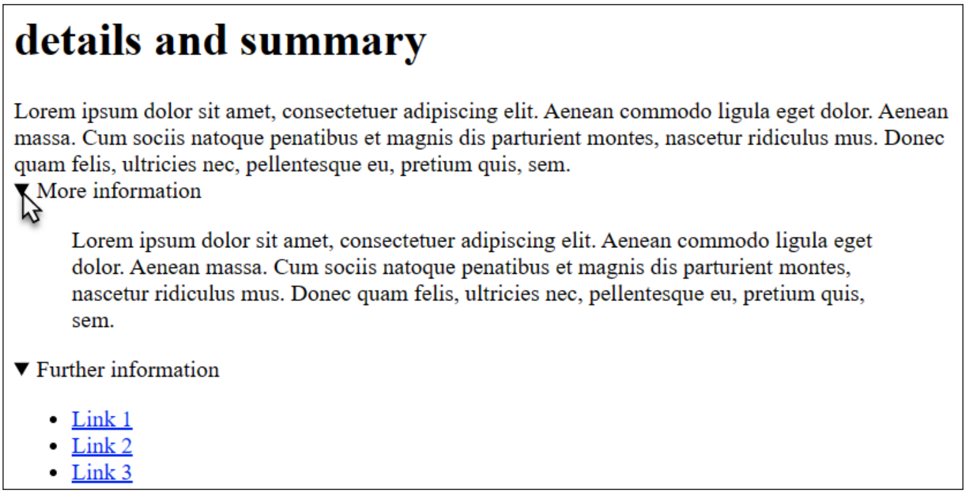
Let's look at a simple example:

```
...

   Lorem ipsum dolor sit amet ...
   <details>
     <summary>More information</summary>
     <blockquote>Lorem ipsum dolor ... </blockquote>
     </details>
   <details open>
     <summary>Further information</summary>
     <ul>
       <li><a href="#">Link 1</a></li>
       <li><a href="#">Link 2</a></li>
       <li><a href="#">Link 3</a></li>
     </ul>
    </details>

...
```

Listing 7.27 /examples/chapter007/7_7_1/index.html

We've written the section for expanding and collapsing between `<details>` and `</details>`. You can specify the clickable area as a heading, which is always displayed between `<summary>` and `</summary>`. Clicking on this `summary` element will expand and collapse the content. If you don't use a `summary` element, the default heading is usually just **Details**. In the example, you can therefore use the headings **More information** and **Further information** to expand and collapse the content hidden by them up to the end of the `details` element.

In the example, I also used the standalone HTML attribute `open` for the second `details` element, which you can use to specify that the content of this `details` element is expanded upon loading of the web page. By default, the content of the `details` element is always collapsed. All current web browsers can handle the `details` element. If the `details` element isn't supported by a web browser, everything will be completely expanded, and no further interaction will be displayed.



details and summary

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem.

▼ More information

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem.

▼ Further information

- Link 1
- Link 2
- Link 3

Figure 7.31 Expandable and Collapsible Content with <details> and <summary>

7.7.2 A Dialog Box via <dialog>

Dialog boxes or on-the-fly popups have been and still are primarily implemented using `div` elements. But there's also a separate HTML element available for this purpose: `<dialog>`. Let's look at a simple example:

```
...
  <dialog id="dialog1" open>
    <p>Exit dialog?<br>
      <input type="button" value="Close"
             onclick="document.getElementById('dialog1').close()">
    </p>
  </dialog>
...
```

Listing 7.28 /examples/chapter007/7_7_2/index.html

You can define a dialog in the space between `<dialog>` and `</dialog>`. The `open` attribute allows you to specify that this dialog gets displayed immediately upon loading. By default, without `open`, the dialog doesn't display. Otherwise, the `dialog` element has no further attributes and only becomes meaningful in interaction with JavaScript. In the example, the dialog box can be closed via a button when clicking (`onclick`). For this purpose, we've addressed the ID name of the dialog with `getElementById('dialog1')` and closed it with `close()`.

In addition to `close()`, you can also use `show()` here to display the dialog, or you can use `showModal()` to display it while everything else is grayed out. Usually, these JavaScript methods are activated by other elements because by default a dialog box without the `open` attribute isn't visible at all at first. But let's not get too much into JavaScript at this point.

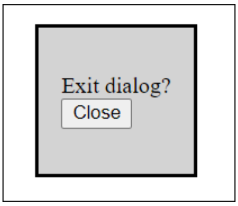


Figure 7.32 A Simple Dialog Box with the HTML Element <dialog>

The Safari browser can't handle the `dialog` element yet. In Firefox, the element must be optionally enabled. Web browsers that can't handle the `dialog` element display the full content, and interaction such as closing the dialog isn't possible.

7.8 Summary

Admittedly, the chapter about HTML forms provides a lot of information because here several worlds collide: HTML, CSS, often also JavaScript, and a script language such as PHP.

You've learned the following in this chapter:

- You know how to create basic forms.
- You're now familiar with the individual HTML input fields for forms.
- Likewise, you're now familiar with the special types of HTML input fields for forms that can make life easier for you and users.
- Thanks to the HTML attributes, many things can be implemented more easily to guide users when filling out forms. In addition, you also know how to do without JavaScript in the future to check HTML input fields.
- Based on a simple example, you've learned how the data in an HTML form is transmitted from the web browser to the web server and processed there via a PHP script.
- How to use interactive HTML elements.

Chapter 8

Introduction to Cascading Style Sheets

In this first chapter on Cascading Style Sheets (CSS), I'll explain what CSS is exactly, as well as the principle of applying CSS based on some simple examples.

In Figure 8.1, you can see a simple basic diagram showing the components that make up a simple and ordinary modern website. Above all, the content is simple. To prepare this content for the web, you can use HTML, which gives the content a semantic meaning by means of HTML elements. In addition, you can use other media files such as images or videos, which are also embedded in the HTML document. Furthermore, more and more often, you can find various scripts for special actions. Finally, the particular focus of this figure and of the entire chapter, is the stylesheet. Taken together, all these basic components form a simple but modern website that gets displayed in a web browser.

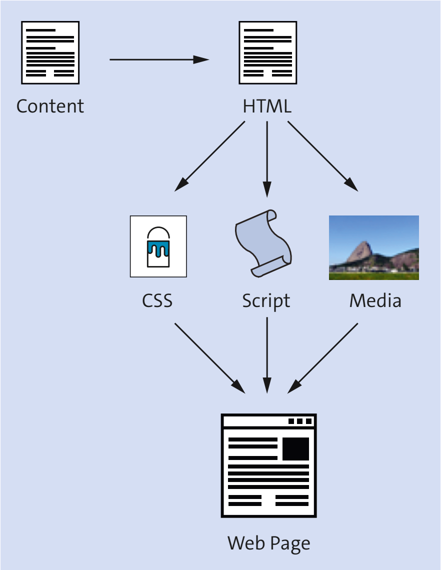


Figure 8.1 The Basic Composition of the Components of a Simple Website

The predominant role of CSS (or stylesheets) in creating websites is the appearance. With CSS, you can create rules for how the content of an HTML element should be displayed. At this point, it pays off to have semantically meaningful notations of the individual HTML elements in the HTML document.

CSS supports the separation of a document's structure and design. Ideally, the presenting aspects should be completely separated from the content of the website, which means the following:

- HTML defines the meaning or semantics of the content.
- CSS defines the presentation of the content.

Without HTML, the content can't be used by web browsers, and without CSS, the content is less beautiful.

Barrier-Free Access and CSS

Thanks to the separation of the content of the HTML document from the design with CSS, the barrier-free access for people with limited abilities can be facilitated.

In this introductory chapter to CSS, I'll cover the following topics:

- The history of CSS
- The principle by which CSS works in practice
- The options involved in using CSS in an HTML document

8.1 The Story of CSS

Because CSS is now the standard stylesheet language for websites, I'd like to describe its genesis here in a fast-forward mode:

1. **The first version with CSS level 1**
   In the early days, there were several similar approaches besides what is now known as CSS, but the inventors of the original CSS, Håkon Wium Lie and Bert Bos, were the first to make the idea public. They were just in the right place at the right time. In 1995, the W3C became aware of CSS during a presentation, and at the end of 1996, the *CSS Level 1 Recommendation* was published. The first version was mainly about the design of fonts and color.
2. **The second version with CSS level 2**
   The next version was released in 1998. Because there were some inconsistencies and CSS Level 2 often caused problems when used on the web due to different web browsers implementing many things incompletely or incorrectly, this version was revised in 2002 with an intermediate version—CSS Level 2 Revision 1—in which some of these defects were fixed or deleted. It took until 2011 before CSS 2.1 was published as a *Recommendation*. This new version included the positioning of elements.
3. **CSS3**
   The third version of CSS has been in the works since 2000. Unlike the Level 2 version, they no longer used a single specification, but instead used CSS3 to split the various

features into different modules. Each module adds new capabilities and extends the features defined in CSS 2.1 with it—keeping everything backward compatible.

Today, CSS no longer has a version number and consists of numerous modules that are developed independently at different paces. CSS3 is really just a term for the modules that were added after CSS 2.1. The individual modules, however, do have a version number. As a result, a CSS4 version will probably never exist.

Overview of the CSS Specification

An overview of all modules in progress can be found at *www.w3.org/Style/CSS/current-work*, which is maintained by co-inventor Bert Bos. A snapshot of the current state of CSS can be found periodically at *http://w3.org/TR/CSS*.

8.2 The Basic Principle of Using CSS

To use the CSS principle so that it makes sense, you should first create the HTML document with logical and semantic HTML elements. You've already learned in detail how to create a proper HTML document with HTML elements in the previous chapters. With CSS, you can use rules for the individual HTML elements to determine their appearance. As an example (see Figure 8.2), we'll use a `header` element, which has been written as shown here in the HTML document, *index.html*:

```
...
  <header>
  <h1>My cooking blog</h1>
  <p>A blog with delicious recipes ...</p>
  </header>
...
```

Listing 8.1 /examples/chapter008/8_2/index.html

You'll assign a new style to this HTML element using CSS, which will determine the formatting and appearance of the `header` element. In this example, you can find this formatting rule described in the external *style.css* file as follows:

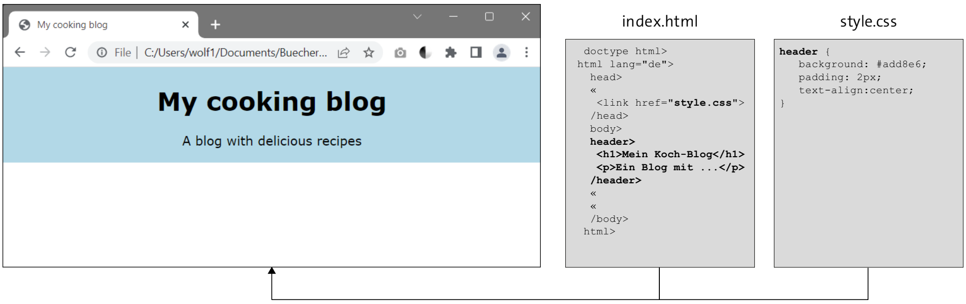
```
/* File: style.css */
...
header {
  background: #add8e6;
  padding: 2px;
  text-align:center;
}
...
```

Listing 8.2 /examples/chapter008/8_2/style.css

At this point, it isn't important to understand what is written in *style.css*. The only important thing is that you can see here how to apply *CSS rules* to an HTML element. In the example, you create the rule in the *style.css* file with the `header` *selector* (i.e., type selector) and the *declarations* between the curly brackets. For this CSS rule to really affect *index.html* and the `header` elements it contains, the HTML document must know where the CSS file (*style.css*) was stored. In the example, you inform the web browser about this using the `link` element.

Using this CSS rule in the *style.css* file for the `header` element(s) in the *index.html* file, you specify that for the content between the HTML tags `<header>` and `</header>`, the background color should be blue (`#add8e6`). The inner spacing (`padding`) is 2 pixels, and the text alignment (`text-align`) is `center`. But as I said, these declarations aren't yet really important in this example. Figure 8.2 demonstrates this process.

If you apply the principle of CSS rules with selectors and declarations to several HTML elements, the outer appearance will change significantly, as you can see in Figure 8.3. This example shown here can be found in /examples/chapter008/8_2/ with *index.html* and *style.css*.



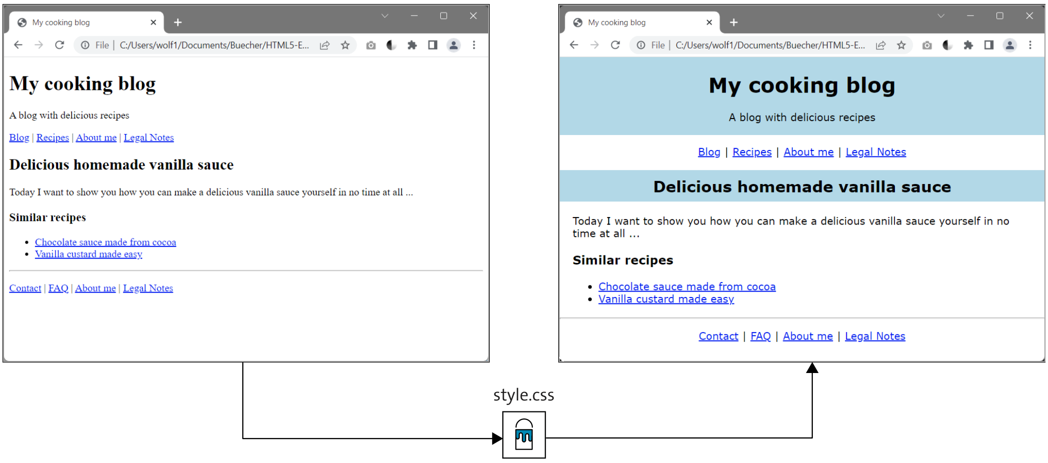
My cooking blog

A blog with delicious recipes

```
 doctype html>
html lang="de">
  head>
  «
   <link href="style.css">
  /head>
  body>
  header>
   <h1>Mein Koch-Blog</h1>
   <p>Ein Blog mit ...</p>
  /header>
  «
  «
  /body>
 html>
```

```
header {
   background: #add8e6;
   padding: 2px;
   text-align:center;
}
```

Figure 8.2 A CSS Rule Is Defined with a Selector and the Declarations It Contains


My cooking blog
File | C:/Users/wolf1/Documents/Buecher/HTML5-E...
My cooking blog
File | C:/Users/wolf1/Documents/Buecher/HTML5-E...
style.css

My cooking blog

A blog with delicious recipes

Blog | Recipes | About me | Legal Notes

Delicious homemade vanilla sauce

Today I want to show you how you can make a delicious vanilla sauce yourself in no time at all ...

Similar recipes

- Chocolate sauce made from cocoa
- Vanilla custard made easy

Contact | FAQ | About me | Legal Notes

My cooking blog

A blog with delicious recipes

Blog | Recipes | About me | Legal Notes

Delicious homemade vanilla sauce

Today I want to show you how you can make a delicious vanilla sauce yourself in no time at all ...

Similar recipes

- Chocolate sauce made from cocoa
- Vanilla custard made easy

Contact | FAQ | About me | Legal Notes

Figure 8.3 Several CSS Rules Have Been Applied to the Individual HTML Elements

8.2.1 Structure of a CSS Rule

As you've already learned, you can define a CSS rule with a selector and a declaration. Selectors are an essential building block of CSS, and there are many different types of them. In this section, I'm not going to cover these selectors in detail yet, but you'll learn how you can construct such a CSS rule in general:

- **Selectors**
  You can use the selector to specify the HTML element to which the CSS rule should be applied. It's also possible to apply a rule to multiple HTML elements by separating the individual HTML elements with commas:

  ```
  h1, h2, h3, p { color: blue; }
  ```

  This sets the CSS rule that the font `color` is `blue` for the HTML elements `h1`, `h2`, `h3`, and `p` at the same time.
- **Declarations**
  You can use the declarations to specify how you want to format the HTML elements selected via the selector. The declaration itself also consists of two parts, a *property* and a *value*. The property is separated from the value by a colon.

In Figure 8.4, you can see the structure and the individual components of a CSS rule.

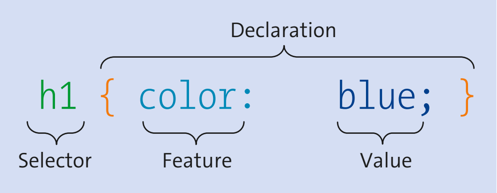


Figure 8.4 Structure of a Simple CSS Rule (CSS Statement)

8.2.2 Declaring a Selector

The declaration inside the curly brackets of a CSS rule consists of at least one *property* and one value, with a colon between the property and the value. If you want to use several such property-value pairs, you must close each pair with a semicolon:

```
h1 {
  font-family: "Arial";
  color: red;
  text-align: center;
}

h2, h3 {
  font-family: "Courier";
  color: blue;
}
```

Here, you specify that all `h1` elements are displayed in Arial font, in red text color, and centered. Furthermore, you set a CSS rule for `h2` and `h3` elements, which are to be displayed in Courier font and in blue color. You can choose the order of the statements as you like. For example, you can also define the color first and then the font.

You could omit the semicolon from the last (or only) property-value pair, but common practice has shown that you usually add more CSS features to a CSS rule, and people tend to forget the omitted semicolon to separate two property-value pairs. A missing semicolon between two property-value pairs is an error.

Let's return to the two components of a CSS declaration:

- **Properties**
  You specify a CSS feature (e.g., color, font, alignment) that you want to change for the HTML element selected with the selector. CSS has a tremendous number of features, many of which you'll get to know throughout the book.
- **Values**
  You specify the value for the CSS feature used. Again, what you can use here depends on the CSS feature you're using. For example, if the property is `color`, you can specify the value of a color. In addition to CSS features, you'll also learn about many different possible value specifications.

8.2.3 Using Comments for CSS Code

If you maintain more CSS code and larger web projects, you should comment your CSS code as clearly as possible so that even after a few weeks, you'll still know what it is and what you've done there. You can introduce a comment via /* and close it with */. Everything in between (including line breaks) will be ignored by the web browser, for example:

```
/* Creates a circle */
/* Warning! Does not work with IE8 or older */
.circle {
  height: 50px;
  width: 50px;
  border-radius: 50px;
}
```

Such comments are also useful if you divide your stylesheets into individual sections to be able to orient yourself more quickly in the CSS code, for example:

```
/*------------------------------------*/
/*        Header and footer area      */
/*------------------------------------*/
...
```

```
CSS statements for header and footer
...
/*-----------------------------------*/
/*             Contents              */
/*-----------------------------------*/
...
CSS statements for the main content
...
```

8.2.4 A Few Notes on Formatting CSS Code

While formatting CSS code is a matter of personal taste, I'll nevertheless give you a few pointers on this. At the least, if you want to change or fix something, you might have some problems finding what you're looking for quickly with the following formatting of the CSS code:

```
/* All right, but very confusing */
h2,h3{font-family:"Courier";color:blue;text-align:center;}
```

A general recommendation for this is to use an extra line for each declaration and indent it. You should put the closing bracket on a separate line. The following is much better to read than the previously shown formatting of the CSS code:

```
/* Much better to read */
h2,
h3  {
   font-family: "Courier";
   color: blue;
   text-align: center;
}
```

For CSS rules with only one declaration, on the other hand, you could write everything in one line:

```
h1 { color: blue; }
```

As mentioned, everyone has their own style, which they develop and continue to use over time.

8.3 Integrating CSS into HTML

There are several ways to associate CSS style statements with an HTML document. Strictly speaking, you have three options at your disposal:

- **Inline style**
  You write the CSS style specifications directly in the HTML element.
- **Internal stylesheet**
  You specify the style statement internally in the header of the HTML document in the `style` element.
- **External stylesheet**
  You use an external stylesheet file and link it to the HTML document.

The following sections describe these three options in greater detail.

8.3.1 Style Statements Directly in the HTML Tag Using the HTML Attribute "style"

This is the worst way to make a CSS statement: writing the style statement(s) directly in the opening HTML tag using the global HTML attribute `style`. Within the `style` attribute, the same syntax and grammar applies to the value assignment as I described in Section 8.2.2. Concerning the logic, a selector isn't needed because the HTML element is already specified in the opening tag to which this CSS rule is applied. Here's a code snippet that shows how you can use such style statements within an opening HTML tag:

```
...
  <header style="background: #add8e6; padding: 2px;
                 text-align: center;">
  <h1 style="font-family: Verdana;">My cooking blog</h1>
  <p style="font-family: Verdana;">A blog with delicious recipes...</p>
  </header>
  <nav style="text-align: center;">
  <p style="font-family: Verdana;">
    <a href="#">blog</a> | <a href="#">recipes</a> |
    <a href="#">About me</a> | <a href="#">Legal notes</a>
  </p>
  </nav>
...
```

Listing 8.3 /examples/chapter008/8_3_1/index.html

Right away, we can see that in this small example, you'll lose the overview if you insert the style statements of CSS directly into the HTML element (also referred to as *inline styles*). Such a style statement within an HTML tag applies only to the HTML element in which that style statement was written. For example, statements have to be repeated as shown here with the `p` element with `font-family`, and if you keep adding style after style in this way and make yet another mistake, it becomes very tedious and usually even more error-prone. If you do this with all your web pages, you'll have to change all web pages when you want to implement any changes. In addition, you would even have to customize every styled element in every single web page.

Can Style Statements Be Used Directly in the HTML Tag So That It Makes Sense?

The only—but also not very convincing—argument for the possibility to use a style directly in an HTML tag could be for testing or demonstration purposes, just to quickly see what something looks like with CSS, or maybe if you want to apply a style only at one specific place in the HTML document. Nevertheless, in most cases, you're better off with the other options of including CSS in HTML because that reduces the maintenance effort enormously and provides you with extreme flexibility.

8.3.2 Style Statements in the Document Head Using the HTML Element <style>

The second option to write CSS style statements is in the HTML document head between `<head>` and `</head>`. When doing this, you mark the area for the style statements with the CSS rules with `<style>` and `</style>`.

Let's take a look at a simple example:

```
<!doctype html>
<html lang="en">
  <head>
    <title>My cooking blog</title>
    <meta charset="UTF-8">
    <style>

       body { margin: 0px; }
       h1 {
         font-family: "Verdana", "Geneva";
         font-size: 200%;
         text-align: center;
       }
       p { font-family: "Verdana", "Geneva"; }
       ...

 <style>
  </head>
  <body>
  ...
  </body>
</html>
```

Listing 8.4 /examples/chapter008/8_3_2/index.html

The CSS rules with style statements that you write between `<style>` and `</style>` apply to the entire HTML document and thus to each HTML element for which you've written a selector. Here, you also can write several of those `style` areas within the HTML document head.

Meaningful Use for Style Statements in the HTML Document Head

This variant is often used to learn CSS because you have everything in one file, which makes it clearer for such purposes. In practice, this option can still be useful if you want to apply or restrict some CSS rules to only one HTML document. However, this method is less suitable for large projects, because you'd have to search and revise each document when changes or errors occur.

8.3.3 Integrating Style Statements from an External CSS File Using <link>

In most cases, when developing more extensive websites, the complete separation of HTML and CSS into separate files is probably the best solution. This is the only way to ensure that the layout is consistent for a larger web project. It also means that you usually have only one CSS file for several HTML documents, which you include in the HTML document with the `link` element within the HTML document head.

Type Description of a CSS File

Like an HTML document, a CSS file is a plain text file with the file extension *.css* (e.g., *mystyle.css*).

Thus, if you combine the CSS rules for formatting in an external CSS file, you only need to make changes in the one central location so that they apply to all other HTML documents that have integrated that CSS file and use the CSS rules.

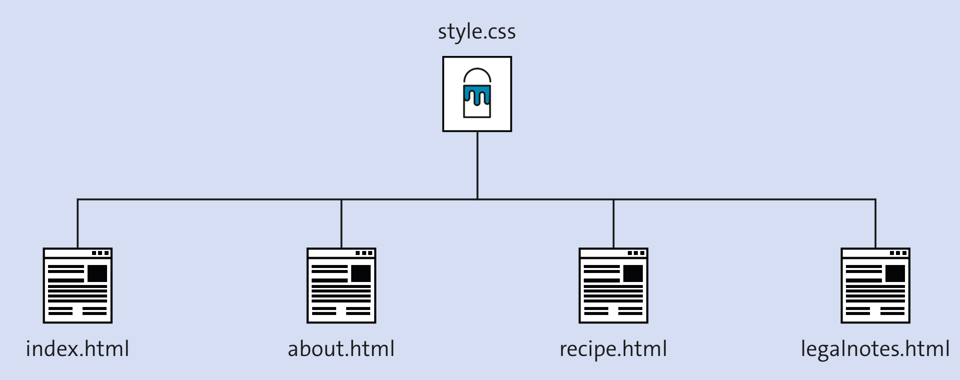


Figure 8.5 By Consolidating CSS Rules in One Place, Design Changes Are Much Easier and Faster to Implement

The following code snippet shows you how to include the CSS file with the HTML element `<link>` in the HTML document:

```
<!doctype html>
<html lang="en">
  <head>
    <title>My cooking blog</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
  ...
  </body>
</html>
```

Listing 8.5 /examples/chapter008/8_3_3/index.html

Although I've already described the `link` element in Chapter 3, Section 3.5, a few sentences should still be added here. You can use the `href` attribute to reference the desired CSS file (here, *style.css*) to be included in the HTML document. In the example, the file is located in the same directory as the HTML document, *index.html*. If this file is located in another directory or even on another server, you must add the corresponding path or URL. Via `rel`, you can write the relationship type of the element, which, with the attribute value `stylesheet`, means that just a stylesheet is to be included.

The code for the CSS file can be found in */examples/chapter008/8_3_3/style.css*. The final result corresponds to Figure 8.3, shown earlier.

8.3.4 Combining CSS Rules in the Head Section and in External CSS Files

The question you're probably asking yourself is which CSS rule takes precedence when you reference an external CSS file with the `link` element while using a range between `<style>` and `</style>`. This is absolutely legitimate and can be used to combine the CSS rules. Let's take a look at a simple example:

```
<!doctype html>
<html lang="en">
  <head>
    <title>My cooking blog</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="style.css">
    <style>
      p { text-align: center; }
    </style>
  </head>
  <body>
  <h1>A headline</h1>
  <p style="text-align: left;">First paragraph text ...</p>
  <p>Second paragraph text ...</p>

  </body>
</html>
```

Listing 8.6 /examples/chapter008/8_3_4/index.html

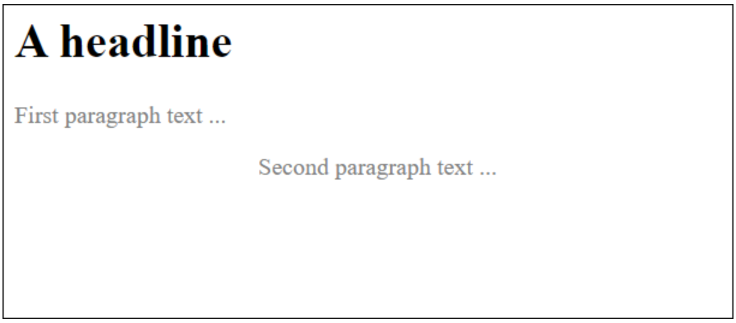
The external CSS file *style.css* contains only the following:

```
/* File: style.css */
p { text-align: right; color: gray; }
```

Listing 8.7 /examples/chapter008/8_3_4/style.css

If there's a conflict of the same rules, the last rule written will take precedence. For example, if the `link` element comes after the `style` section, the CSS rule from the external CSS file will take precedence. However, if you still write a style statement in the opening HTML tag (as an inline style), then the CSS rule in the opening HTML tag takes precedence. CSS features that don't overlap, such as the text color (`gray`) in the *style.css* file of the example, are combined with the existing CSS rules.

For this reason, in the example, `p` elements are displayed in gray color because there was no overlap with the `color` property. The first paragraph text, on the other hand, is left-aligned because the most local CSS rule in the HTML tag for the `p` element was written there as an inline style. The second paragraph, in turn, uses center alignment because the more local CSS rule for the `p` element was defined in the `style` section the document head. The specification of `text-align: right;` in the stylesheet file *style.css*, on the other hand, doesn't get executed at all because there are more local CSS rules containing the `text-align` property for both `p` elements.



A headline

First paragraph text ...

Second paragraph text ...

Figure 8.6 Result of Combining Style Statements within an HTML Tag in the <style> Section of the Document Head and in a Separate CSS File

You can even reference multiple CSS files using the `link` element. In that case, too, for the same HTML elements with different declarations of CSS features, those files will be combined. If a conflict exists between two CSS features that are the same but have different values for an HTML element, the style sheet that's integrated at a later time overrides the specification of the previously integrated one.

Cascade

Because style statements can be integrated and combined in different ways, there must be a rule for this that decides which property takes precedence when there are competing CSS rules. The problem is solved in CSS by having the cascade calculate a weighting (a points system) for the rules and properties that determines the format applied to an element.

8.3.5 Recommendation: You Should Separate HTML and CSS

The previous sections have described the options you have for using CSS for websites. In the previous section, you may already have noticed how confusing it can get when you distribute CSS all over the HTML document. It's obvious that when mixing these options, the overview is gone. In our example, the listed examples and the combination of different ways to use CSS served only to demonstrate what is allowed and theoretically possible. In practice, it's recommended to write the CSS code in a separate file and include it in the HTML document head via the `link` element. Some of the advantages of using a central CSS file (refer to Figure 8.5) are listed here:

- You have a consistent layout even for larger projects. All format properties that you defined in the central CSS file apply to all HTML documents in which this CSS file is included.
- The effort for maintenance or design changes is considerably reduced because this work will be limited to the single CSS file, and the changes of the central file will immediately be available for all HTML documents.
- The HTML documents will become smaller in size because they consist only of HTML. This also reduces loading time because the central stylesheet file only needs to be downloaded once and can then get cached in the web browser.

8.3.6 Testing Alternate Stylesheets during Development

Using the global `title` attribute, you can set up alternate style sheets within the `link` or `style` elements. This could be useful for development work in a team, for example, when you want to compare and test different CSS themes. A simple example of this is shown in Listing 8.8.

```
...<head>
    <title>Alternate CSS</title>
    <meta charset="UTF-8" />
    <link rel="stylesheet" href="normal.css" title="Bright mode" />
    <link rel="alternate stylesheet" href="dark.css" title="Dark mode" />
    <link rel="alternate stylesheet" href="light.css" title="Light" />
</head>
...
```

Listing 8.8 /examples/chapter008/8_3_5/index.html

In this example, three stylesheets with different color schemes are provided. Because all three of them contain the `title` attribute, the first element (`title="Bright mode"`) is used for rendering. The other stylesheets should be selectable by the web browser as an alternative. For example, in Firefox, you can select an alternate stylesheet specified in the `title` attribute from the **View** menu in the **Web Page Style** submenu.

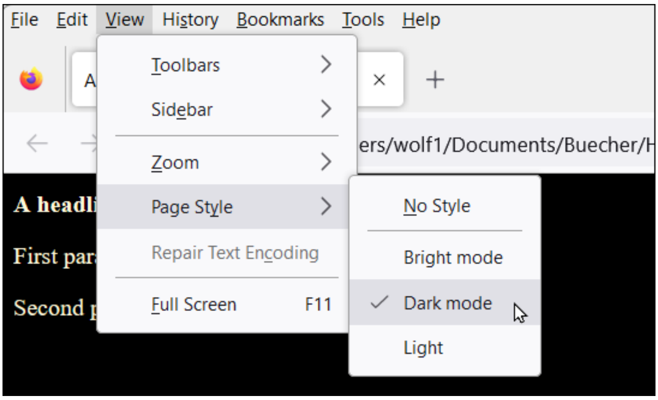

File
Edit
View
History
Bookmarks
Tools
Help
A
ers/wolf1/Documents/Buecher/H
Toolbars
Sidebar
Zoom
Page Style
Repair Text Encoding
Full Screen F11
No Style
Bright mode
✓ Dark mode
Light
A headli
First par
Second p

Figure 8.7 Selecting an Alternate Stylesheet in Firefox

As mentioned earlier, this feature can be very handy when you work in a team of developers. However, because it isn't supported by all web browsers, you shouldn't use it for public web projects. Firefox has been able to handle this function since version 3. For other web browsers, however, you need an extension. If you use multiple alternate stylesheets, the following rules apply:

- If `alternate` hasn't been used with the `rel` attribute and no `title` attribute has been used, the stylesheet will be preferred.
- If a style has the `title` attribute and hasn't been marked as `alternate`, it will be used as the default stylesheet.
- If a style has the `title` attribute and has been marked as `alternate`, it will be listed in the menu list.

8.3.7 Integrating Style Statements from an External CSS File Using “@import”

Besides the HTML syntax for integrating an external CSS file in an HTML document using the `link` element, there’s a second possibility in CSS: the `@import` rule. Here’s a simple example of how you can reference an external CSS file with this rule:

```
...
  <head>
    <title>My cooking blog</title>
    <meta charset="UTF-8">
    <style>

      @import url("style.css");

    </style>
  </head>
...
```

Listing 8.9 /examples/chapter008/8_3_6/index.html

The `@import` rule must also be written in the HTML document head between `<head>` and `</head>`. To be more precise, it must be written in the `style` section between `<style>` and `</style>`. You can use `@import url("style.css");` to include the CSS file in the HTML document. Again, the stylesheet file is assumed to be in the same directory as the HTML document.

In practice, this example makes little sense compared to importing via the `link` element and is only meant to illustrate the use of the `@import` rule. More often, you’ll include a stylesheet via `<link>` and import other stylesheets from that stylesheet via the `@import` rule.

The important aspect about this `@import` rule is that you write it at the beginning of the `style` section. There must be no other CSS statement before the `@import` rule in the `style` section. After that, on the other hand, you can write CSS rules as you like. You can also include other external CSS files with the `@import` rule.

8.3.8 Media-Specific Stylesheets for Specific Output Devices

If you want to set stylesheets for a specific output medium, you can do so by using the `media` attribute in the `link` element. This gives you the option, for example, to apply a stylesheet only to certain output media. The following example demonstrates how you can specify a media-specific stylesheet for the screen and a different stylesheet for the printer:

```
...
  <head>
    <title>My cooking blog</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css"
     media="screen" href="style.css">
    <link rel="stylesheet" type="text/css"
     media="print" href="print.css">
  </head>
...
```

Listing 8.10 /examples/chapter008/8_3_7/index.html

If the output device is a screen (`media="screen"`), then the HTML document is formatted using *style.css*. If, on the other hand, the output medium is a printer (`media="print"`), the document will be formatted using *print.css*. Alternatively, you can use the `@import` rule for this instead of linking the CSS files via the `link` element:

```
...
  <head>
    <title>My cooking blog</title>
    <meta charset="UTF-8">
    <style>

      @import url("style.css") screen;
      @import url("print.css") print;

    <style>
  </head>
...
```

Table 8.1 provides an overview of the media-specific attribute values you can use to assign media-specific stylesheets.

Attribute Value	Output Device
`all`	All output devices (default value)
`print`	Printer
`screen`	Screen-oriented output devices

Table 8.1 Media-Specific Output Devices for Stylesheets That Can Be Assigned to the "media" Attribute

Apart from that, there are other media types or device classes such as `aural`, `braille`, `embossed`, `handheld`, `projection`, `speech`, `tty` or `tv`, that have been classified as deprecated since Media Queries Level 4, so you should refrain from using them. However, you can also assign multiple values separated by commas. If you don't specify an attribute value, the attribute value `all` will be used, which the stylesheet is used with regardless of the output medium.

"@media" Statements within a Stylesheet

You don't need to swap out the media-specific stylesheet statements to a separate file; you can also use the `@media` rule within a stylesheet to create individual CSS rules for specific media. Here's an example of how you can optimize the font size specifically for printing:

```
p { font-size: 1.6em; }
@media print {
  p { font-size: 10pt; }
}
```

For the `p` element, you use `1.6em` as the font size with `em` as the unit of measure for a relative font size for all media except print. For printing, on the other hand, you can use the `@media` rule and `print` to use a `pt` (= point) unit of measure suitable for printing, with `10pt` as the font size. I'll deal with the topic of units of measure for fonts separately because it's not relevant at this point. This is just about using the `@media` rule.

8.3.9 Media-Specific Stylesheets with CSS

In addition to the options just shown, there are also media-specific stylesheets (also called media queries) that play a key role in responsive web design. For this purpose, logical operators (`and`, `not`) have been introduced, allowing you to perform queries about a wide variety of media properties, such as usable screen width or screen orientation (portrait/landscape for tablets). For example, if you want to provide a special stylesheet for a 1,080-pixel screen, you can do so as follows:

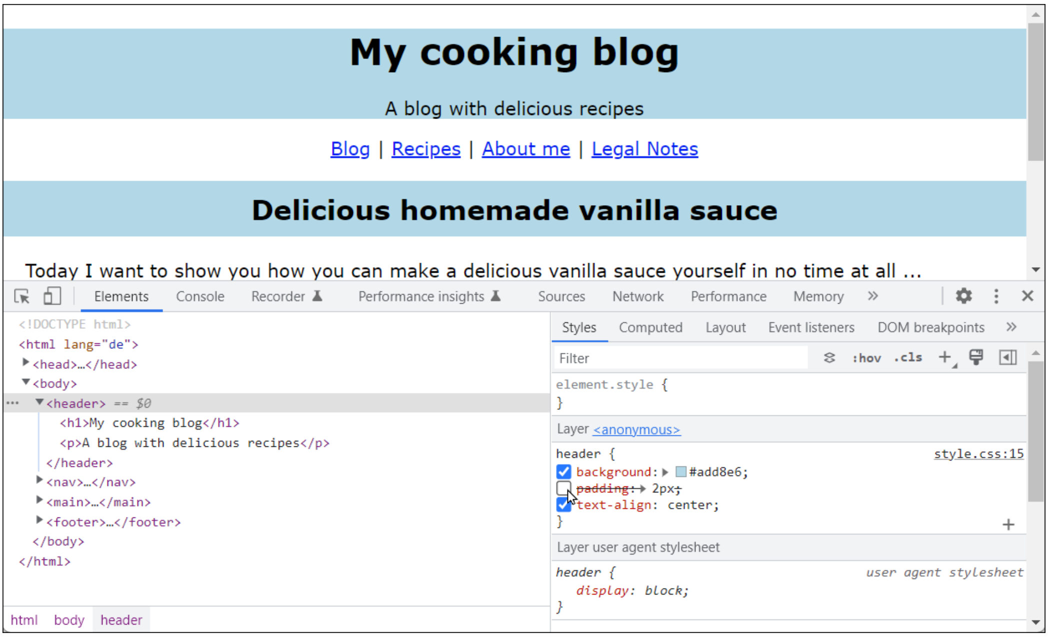
```
<link rel="stylesheet" media="screen and (min-width: 1080px)"
      href="style1080.css">
```

This will include the CSS file *style1080.css* in the HTML document if the media has a `screen` and `and` that screen is at least 1,080 pixels wide (`min-width: 1080px`). Many more such media properties are available for this, just like `min-width` here. But those media queries will be described separately in this book. Before that, you'll be introduced to the basics of CSS.

8.4 Analyzing CSS in the Web Browser

The developer tools provided by each web browser represent good learning and support tools. Besides analyzing HTML, they also enable you to analyze the CSS, which is particularly useful if you want to learn how other websites have designed a particular element. In almost all web browsers you can access the developer tools by pressing the `Ctrl` + `Shift` + `I` shortcut.

If, for example, you want to examine a styled HTML element, you can select it as shown in Figure 8.8 with `<header>`; for this, you then get the corresponding style (here, with **Styles**) displayed on the right side. The highlight is that you can (de)activate the style there on a test basis or change the values. The changes are only visual as the files remain untouched. The value of **user agent stylesheet** is the fixed stylesheet of the web browser.



My cooking blog

A blog with delicious recipes

Blog | Recipes | About me | Legal Notes

Delicious homemade vanilla sauce

Today I want to show you how you can make a delicious vanilla sauce yourself in no time at all ...

```
<!DOCTYPE html>
<html lang="de">
 ▸<head>…</head>
 ▾<body>
 ··· ▾<header> == $0
        <h1>My cooking blog</h1>
        <p>A blog with delicious recipes</p>
      </header>
     ▸<nav>…</nav>
     ▸<main>…</main>
     ▸<footer>…</footer>
    </body>
</html>
```

```
element.style {
}
Layer <anonymous>
header {                                   style.css:15
☑ background: ▸ #add8e6;
☐ ~~padding: ▸~~ 2px;
☑ text-align: center;
}
Layer user agent stylesheet
header {                          user agent stylesheet
  display: block;
}
```

Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory
Styles
Computed
Layout
Event listeners
DOM breakpoints
Filter
:hov
.cls
html
body
header

Figure 8.8 The Developer Tools of Web Browsers Are Also Very Useful with Regard to Analyzing and Learning CSS

8.5 Summary

In this chapter, you've learned about the basic principle of CSS. I'll go into the individual details of CSS more comprehensively in the course of the next chapters. You also learned about the three ways to include CSS in HTML.

Chapter 9

The Selectors of CSS

The selectors are an essential and indispensable part of CSS. You'll find a comprehensive overview of the many types of selectors in this chapter and also learn how to use them in practice.

In Chapter 8, Section 8.2.1, you've already briefly and fundamentally learned about the structure of a CSS rule and the use of a simple selector, more precisely, a type selector.

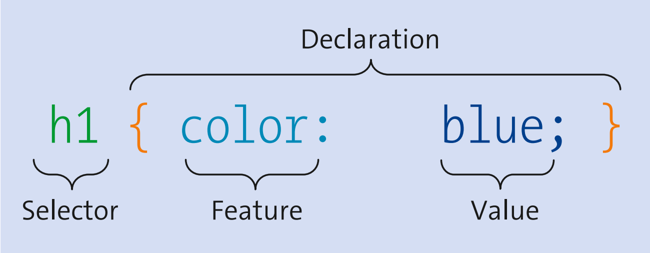


Figure 9.1 The Structure of a Simple CSS Rule with a Selector and a Declaration

But let's briefly summarize again the purpose of a selector. For the format properties to be applied to a specific HTML element, you must define a CSS rule with a selector and a declaration (or multiple declarations). A selector is the part of a CSS rule that comes before the curly brackets ({}) section, as shown in Figure 9.1. You can use the selector to choose the element or elements to which the various format properties declared between the subsequent curly brackets should be applied. You can group several independent selectors for the same declarations if you separate them with commas. A theoretical example follows:

```
Selector {
  CSSProperty1: Value1;
  CSSProperty2: Value2;
  ...
}

Selector_01,
Selector_02,
Selector_03 {
  CSSProperty1: Value1;
  CSSProperty2: Value2;
  ...
}
```

Without such a selector, you wouldn't be able to specify a pattern match with which to address elements in the document tree. The pattern determined by the selector ranges from a simple element name to much more complex patterns. If the pattern matches a particular element, the rule gets applied to the element with the declaration. It's absolutely possible to use no selector or only *, but, in this case, the declarations will be applied to all elements in an HTML document.

CSS Selector Test

If you want to know which selectors are implemented in the web browser you're currently using and which you can use accordingly, you can find a corresponding test at *http://css4-selectors.com/browser-selector-test/*. The level 1 to level 4 selectors are tested.

The goal of this chapter is to introduce you to the different types of selectors so that you can form more complex pattern comparisons using selectors. For this purpose, CSS provides many different selectors, which should be categorized as follows:

- **Simple selectors**
  Simple selectors include the type selector, universal selector (*),class selector (`.class`), ID selector (`#id`), attribute selector, and several pseudo-classes.
- **Combinators**
  Combinators are two selectors concatenated by a greater-than sign (`E > F`; child combinator), the plus sign (`E + F`; adjacent sibling combinator), a tilde sign (`E ~ F`; general sibling combinator), or a space (`E F`; descendant combinator).

A Note on Working through This Chapter

This chapter describes many types of selectors. However, the subject is less spectacular and, in some places, more theoretical. It isn't absolutely necessary to work through this chapter selector by selector. You can also use it just as a reference if a certain selector is used in the book that you aren't yet familiar with, or if you ever get stuck on how to select a certain element or why a different element was selected than expected.

9.1 The Simple Selectors of CSS

This section describes the simple selectors of CSS. These include the universal selector, simple type selector, class selector, ID selector, attribute selector, and pseudo-classes.

9.1.1 Addressing HTML Elements Using the Type Selector

The type selector—sometimes also referred to as an HTML element selector—is the simplest selector, which you've already used several times in this book. Such a type selector addresses the HTML elements directly with the element name. This rule gets applied to all elements of the same type in the HTML document. It's irrelevant where in the HTML document these elements are written, to which class they belong, or which identifier they have.

The following lines show the type selector in use in a separate CSS file:

```
/* Black frame, centered text, 5 pixel distance from top
 */
header,
nav,
footer {
  text-align: center;
  border: 1px solid black;
  margin-top: 5px;
}

/* Gray text
 */
h1,
abbr { color: gray; }

/* Gray dotted frame
 */
p { border: 1px dotted gray; }
```

Listing 9.1 /examples/chapter009/9_1_1/css/style.css

First, for the HTML elements `<header>`, `<nav>`, and `<footer>`, you cause the text to be center-aligned and a black border with a thickness of 1 pixel is to be drawn around it. The distance to the upper next element is 5 pixels for these elements. Then you set the rule that all HTML elements `<h1>` and `<abbr>` are to be displayed in gray color. At the end, each HTML element `<p>` in the HTML document gets a gray dotted frame.

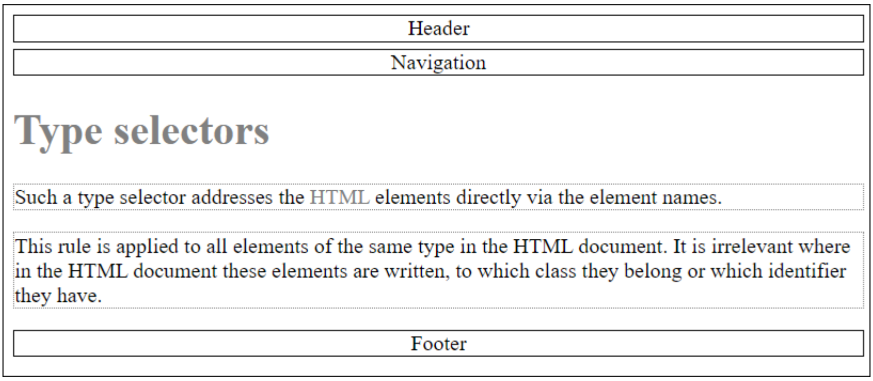
In the following example, you'll add this CSS file, *style.css*, via the `link` element to the HTML document head:

```
...
  <head>
  ...
   <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
```

```
    <header>Header</header>
    <nav>Navigation</nav>
    <main>
      <h1>Type selectors</h1>
      <p>Such a type selector addresses the <abbr>HTML</abbr>
         elements directly via the element names. </p>
      <p> This rule will be applied to all elements of the same
          type in the HTML document. With ... .</p>
    </main>
    <footer>Footer</footer>
  </body>
...
```

Listing 9.2 /examples/chapter009/9_1_1/index.html

As you can see in Figure 9.2, according to the CSS rules, the individual HTML elements are selected using the corresponding type selector and formatted according to the declarations of the rule.



Header

Navigation

Type selectors

Such a type selector addresses the HTML elements directly via the element names.

This rule is applied to all elements of the same type in the HTML document. It is irrelevant where in the HTML document these elements are written, to which class they belong or which identifier they have.

Footer

Figure 9.2 The Individual HTML Elements Were Selected with the Appropriate Type Selector and Formatted with CSS

Selector	Name	Selection	HTML Example
`element {...}`	Type selector	HTML element named `element`	`<element>`

Table 9.1 The Type Selector Has Been Around Since CSS Level 1

At this point, I have a quick note on how you can save yourself some typing. Consider the following theoretical example:

```
h1 {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 160%;
  color: blue;
}

h2 {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 140%;
  color: blue;
}

h3 {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 120%;
  color: blue;
}
```

If you take a closer look at these lines, you'll see that for all three HTML elements for the headings `<h1>`, `<h2>`, and `<h3>`, only the `font-size` property is different. Accordingly, you can shorten these CSS rules by grouping them as follows:

```
h1,
h2,
h3 {
  font-family: Arial, Helvetica, sans-serif
  color: blue;
}

h1 { font-size: 160%; }
h2 { font-size: 140%; }
h3 { font-size: 120%; }
```

First, you group the HTML elements `<h1>`, `<h2>`, and `<h3>` and set the font family and color. Second, you set the font size for each of these HTML elements.

9.1.2 Addressing HTML Elements Using a Specific Class or ID

The type selector is ideal for styling specific HTML elements in an HTML document. For example, if you want to display all HTML elements `<p>` in blue font, you can address these elements quite easily via the type selector `p {color: blue;}`. If you don't want to assign all HTML elements (here again, `<p>` as an example) in the document the same style, you need to be more specific. For this purpose, the class selector or the ID selector are suitable.

Class Selector: Addressing Elements with a Specific “class” value

You can use the class selector to select the HTML elements that belong to a particular class. In HTML, you can assign such a class name using the global HTML attribute `class` for almost all elements, for example:

```
<p class="note">A paragraph text ...</p>
```

Now you can address the HTML element with the class name `note` in CSS via the class selector by placing a dot in front of the class name, as follows:

```
.note { color: red; }
```

Using “class=""” or “id=""”?

This isn’t a crucial question, but nevertheless, especially as a beginner, one stumbles over these two HTML attributes. You can use the `class` attribute if you want to style multiple elements in multiple places in the HTML document with it. The `id` attribute, on the other hand, can be used only once in the HTML document for an element.

You can also use multiple classes at once in HTML elements by separating the individual classes with a space, for example:

```
<p class="bigfont note">...</p>
```

Here, the two classes `note` and `bigfont` are assigned to the `p` element.

Here’s an example that demonstrates the class selectors in use. First the CSS file:

```
...
/* Font family for all p elements, irrespective of the class.
 */
p { font-family: Verdana, Arial; }

/* Style for a note
 */
.note {
  margin-left: 50px;
  border-left: 10px solid green;
  padding-left: 5px;
}

/* Style for a warning
 */
.warning {
  border-left: 10px solid red;
  border-top: 2px solid red;
  border-right: 10px solid red;
  border-bottom: 2px solid red;
  text-align: center;
}

/* Font size to 140%; background color to gray
 */
.headfoot {
  font-size: 140%;
  background: #f5f5f5;
}

/* Font size to 130%
 */
.bigfont { font-size: 130%; }
```

Listing 9.3 /examples/chapter009/9_1_2/css/style.css

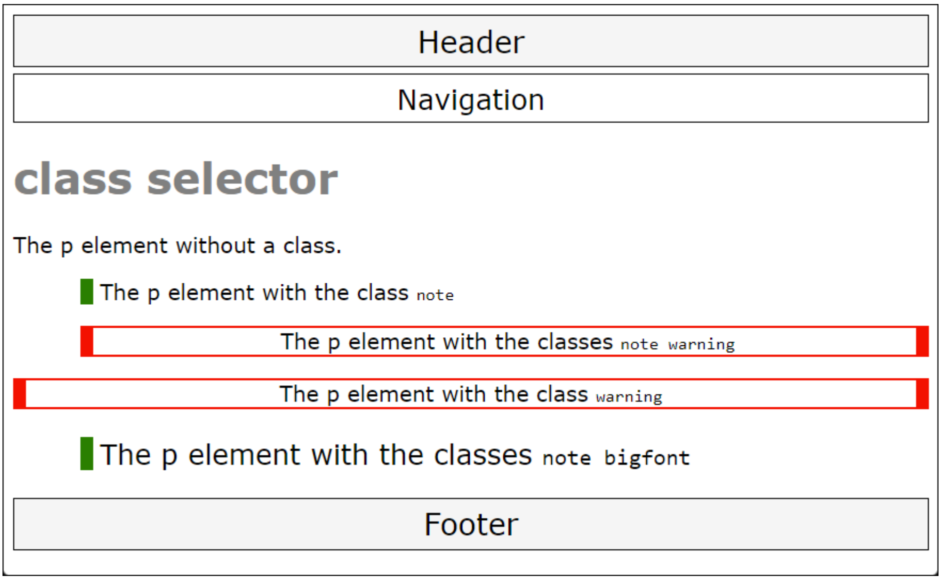
Here's another simple HTML document that uses these class selectors and demonstrates how to use them in practice:

```
...
  <head>
   ...
   <link rel="stylesheet" type="text/css" href="css/style.css">
  </head>
  <body>
    <header class="headfoot">Header</header>
    <nav class="bigfont">Navigation</nav>
    <main>
      <h1>class selector</h1>
        <p>The p element without a class.</p>
        <p class="note">The p element with the class
           <code>note</code></p>
        <p class="note warning">The p element with the
           classes <code>note warning</code></p>
        <p class="warning">The p element with the class
           <code>warning</code></p>
        <p class="note bigfont">The p element with the
           classes <code>note bigfont</code></p>
    </main>
    <footer class="headfoot">Footer</footer>
  </body>
...
```

Listing 9.4 /examples/chapter009/9_1_2/index.html

The first paragraph which contains the `p` element doesn't use a class, so only the type selector `p` with the font family customization is applied to it here. The same font family is used for the other four paragraphs. The second paragraph is formatted with the `.note` class selector, indented by 50 pixels from the left and also adding a 10-pixel green border to the left. The distance to the border is 5 pixels. The third paragraph uses the class selectors `.note` and `.warning`, where everything is formatted quite similarly to the second paragraph. It should be added here that the `.warning` class inherits the properties of `.note`, which thus also affects the indentation. The fourth paragraph, which contains the `.warning` class, on the other hand, contains no indentation because the properties of `.note` aren't known here. In the fifth paragraph, the class selectors `.note` and `.bigfont` were also combined, so that in the example, in addition to `.note`, the font was slightly enlarged.

Similarly, in the example, the HTML elements `<header>` and `<footer>` were selected with the class selector `.header` and given a larger font and a gray background. You can see the result of this example in Figure 9.3.



Header

Navigation

class selector

The p element without a class.

The p element with the class `note`

The p element with the classes `note warning`

The p element with the class `warning`

The p element with the classes `note bigfont`

Footer

Figure 9.3 This Is What the Example /examples/chapter009/9_1_2/index.html Looks Like with the Class Selectors Written in CSS File style.css

You can also associate class selectors with other selectors to specify more precisely which elements should be selected and customized. For example, with regard to the example just shown, */examples/chapter009/9_1_2/css/style.css*, let's suppose you had written the style for the `.warning` class as follows:

```
...
.note.warning {
  border-left: 10px solid red;
  border-top: 2px solid red;
  border-right: 10px solid red;
  border-bottom: 2px solid red;
  text-align: center;
}
...
```

In that case, the fourth paragraph of the example would no longer be formatted because no selector is defined for `<p class="warning">` and `.note.warning` only addresses `<p class="note warning">`.

Likewise, you can connect the class selector to the type selector. For example, you could use a definition such as `p.note` to make sure that only `p` elements containing the class attribute `note` (`<p class="note">`) will be styled. You wouldn't be able to use the `note` class in any other HTML element.

ID Selector: Addressing an Element with a Specific "id" Value

With the ID selector, similar to the class selector, you select the HTML elements to which you've assigned a specific ID. You can assign an ID to an HTML element using the general HTML attribute, `id`. Unlike classes (with the `class` attribute), IDs are always unique elements in an HTML document, so only one element in the entire HTML document can be assigned this ID. The fact that an ID may only occur once is also a reason why the `id attribute` has been preferred over the `class attribute`, especially for `div elements`, to create document-wide unique IDs such as `content`, `header`, and `navigation`.

For this purpose, */examples/chapter009/9_1_2/index.html* has been slightly rewritten: the HTML elements have been removed; instead, the individual sections have been written with the classic `div` element and corresponding `id` attributes. Here's an HTML approach to this:

```
...
  <body>
    <div id="header">Header</div>
    <div id="nav">Navigation</div>
    <div id="main">
      <h1>class selector</h1>
      <p>The p element without a class</p>
      ...
      ...
    </div>
    <div id="footer">Footer</div>
  </body>
...
```

Listing 9.5 /examples/chapter009/9_1_2/index2.html

To create a CSS rule for this, you can use the ID selector with the name of the corresponding `id` attribute value in the HTML document. Such an ID selector must be written with the hash character (#) followed by the ID. The ID selector for `<div id="header">` can thus be written as follows:

```
#header { ... }
```

An ID may occur only once in the document, but this error doesn't prevent the web browser from styling other `id` attributes of the same name with the ID selector as well. A validation immediately reveals such erroneous circumstances.

To illustrate this, here's the complete CSS file where you divide the individual `div` elements into document-wide unique sections based on their `id` attributes:

```
/* Black frame, centered text, 5 pixel distance from top
 */
#header,
#nav,
#footer {
  text-align: center;
  border: 1px solid black;
  margin-top: 5px;
  padding: 5px;
  font-family: Verdana, Arial;
}

/* Font size to 140%; background color to gray
 */
#header,
#footer {
  font-size: 140%;
  background: #f5f5f5;
}

/* 20 pixels distance from all other elements
 */
#main { margin: 20px; }
...
```

Listing 9.6 /examples/chapter009/9_1_2/css/style2.css

The result is exactly the same as the one with the class selector before, shown previously in Figure 9.3.

Like class selectors, you can associate ID selectors with other selectors. For example, you can use `div#header` to connect the type selector to the ID selector. In the example, this would address the `div` element where the `id` equals `header` (`<div id="header">`).

However, because an ID may only be used once anyway, you can omit the type selector in this example and use `#header` as usual.

Mixing the ID Selector with a Class Selector

ID selectors can also be associated with class selectors. It's therefore possible to use a selector in the following form: `.classname#id` and `#id.classname`. However, you should keep in mind that an ID still remains unique across the entire document.

When to Use the Class Selector versus the ID Selector?

The class selector is the *selector with the dot*, while the ID selector is the *selector with the hash*. Likewise, you should keep in mind that while you can assign one or more classes to each HTML element, you can generally assign only one ID. Unlike classes, IDs must be unique in an HTML document. In addition, you can't assign more than one ID to an HTML element.

Permitted Characters for Selectors

For the name of the selector, you can only use uppercase and lowercase letters (`a-z`; `A-Z`), digits (`0-9`), and the hyphen (`-`) as well as the underscore character (_). In addition, the name mustn't begin with a digit.

In your daily work, you should prefer to use class selectors for specific properties or groups such as notes, warnings, or error messages. You can use the ID selectors, on the other hand, to write single or unique main areas of a web page such as `#headarea`, `#mainarea`, or `#footarea`, although the range of HTML elements available for those purposes is absolutely sufficient.

Using Meaningful Class Names and ID Names

The class names and ID names you use should be meaningful and you shouldn't choose any names that reflect the formatting. For example, you should rather avoid a class name such as *redframe* and use *warning* or *error* instead, if this should be the corresponding function. A meaningful name will help you understand the meaning more quickly during a later revision, and it will be easier for you to assign this class or ID name to an HTML element when designing the web pages. In general, styling details should be kept out of the name.

Selector	Name	Selection	HTML Example
`.clname`	Class selector	Element with the `clname` class	`<p class="clname">`
`#elemid`	ID selector	Element with ID `elemid`	`<p id="elemid">`

Table 9.2 Class and ID Selectors

9.1.3 Universal Selector: Addressing All Elements in a Document

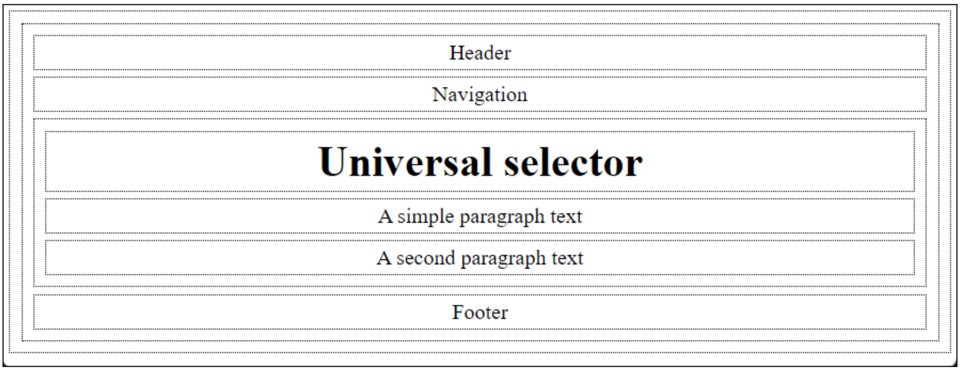
You can use the universal selector to select all HTML elements in a document at once. The universal selector must be written with an asterisk (*).

Let’s take a look at the following example:

```
/* Black dotted frame;
   Distance to the next element: 5 pixels;
   Fill with 3 pixels and center text
 */

* {
  margin: 5px;
  padding: 3px;
  border: 1px dotted black;
  text-align: center;
}
```

Listing 9.7 /examples/chapter009/9_1_3/css/style.css

In this example, the universal selector * draws a black dotted frame around all HTML elements. In addition, the spacing of each HTML element was styled to 5 pixels in all directions, and the interior spacing was styled to 3 pixels. The text is also center aligned. Figure 9.4 shows the universal selector written in *style.css* applied to */examples/chapter009/9_1_3/index.html*.



Header

Navigation

Universal selector

A simple paragraph text

A second paragraph text

Footer

Figure 9.4 The Universal Selector Applied to All Elements Used in the HTML Document

The universal selector can be used not only to select all elements but also to select all elements within an element. Let’s again use the HTML document and the CSS file from which Figure 9.4 was created. Consider the following excerpt of the HTML document:

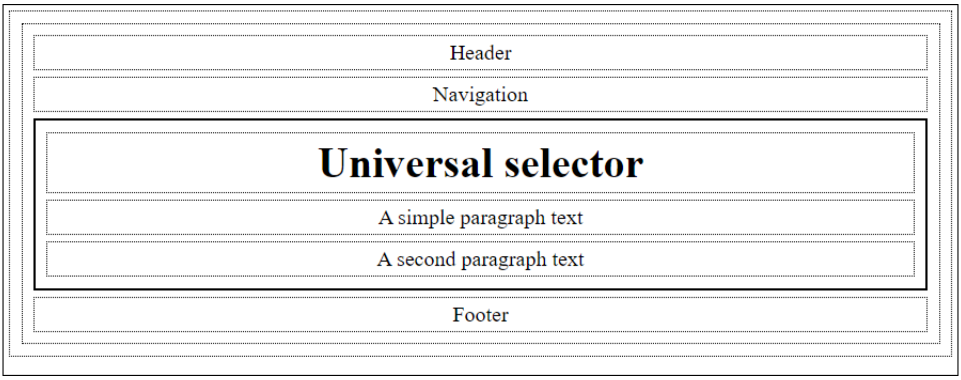
```
...
  <body>
  <header>Header</header>
  <nav>Navigation</nav>
  <main>
    <h1>Universal selector</h1>
    <p>A simple paragraph text</p>
    <p>A second paragraph text</p>
  </main>
  <footer>Footer</footer>
  </body>
...
```

Listing 9.8 /examples/chapter009/9_1_3/index.html

Your main focus should be on the `main` element. If you want to define a CSS rule to put a special border around the `main` element, you can do that using a type selector like the following, as you already know:

```
main { border: 2px solid black; }
```

You can see the result of this type selector in Figure 9.5.



Header

Navigation

Universal selector

A simple paragraph text

A second paragraph text

Footer

Figure 9.5 A Solid Frame with a Thickness of 2 Pixels Was Drawn around HTML Element <main>

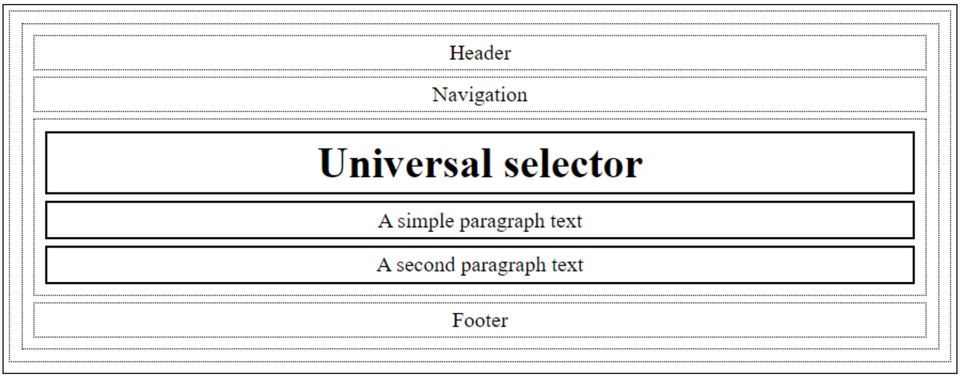
If instead you want to draw the same frame around all HTML elements written inside `<main>`, you can define that using the universal selector as follows:

```
main * { border: 2px solid black; }
```

As you can see in Figure 9.6, thanks to the combination of a type selector and the universal selector, all elements inside the `main` element have now been selected and provided with the frame. Here, I've preempted the descendant selector to demonstrate another example to the universal selector.

Selector	Name	Selection	Example
*	Universal selector	All elements	`<p>`

Table 9.3 Universal Selector



Header

Navigation

Universal selector

A simple paragraph text

A second paragraph text

Footer

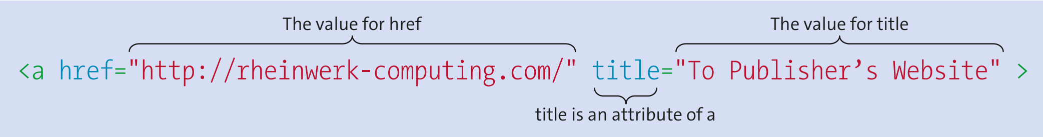
Figure 9.6 A Combination of a Type Selector and the Universal Selector

9.1.4 Addressing Elements Based on Attributes Using the Attribute Selector

The attribute selector can be used to select HTML elements by their HTML attributes. This allows you to check the presence of an attribute or the value it contains. In addition, you also have the possibility of a partial matching of attribute values. Using the HTML element `<a>` in Figure 9.7 as an example, these three attribute selector options are explained in more detail.

If you want to address all HTML elements that have a specific HTML attribute, you merely need to put the attribute name between square brackets:

```
[attributname] { ... }
```



```
<a href="http://rheinwerk-computing.com/" title="To Publisher’s Website" >
```

Figure 9.7 HTML Element `<a>` with Two HTML Attributes

For example, if you want to address all `title` attributes of an HTML document, you can write the attribute selector for it as follows:

```
[title] { border: 1px solid black; }
```

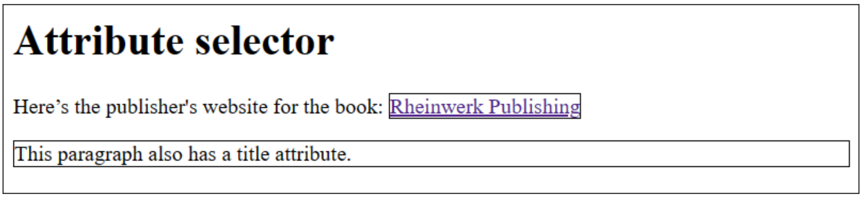
This attribute selector will draw a frame within an HTML document around any HTML element that has a `title` attribute.

The following HTML document is intended to serve as an example and demonstrate the attribute selector in use:

```
...
<h1>Attribute selector</h1>
<p>Here’s the publisher's website for the book:
   <a href="https://www.sap-press.com/"
      title="Publisher’s website">Rheinwerk Publishing</a>
</p>
<p title="A paragraph with title">This paragraph also has
   a title attribute.
</p>
...
```

Listing 9.9 /examples/chapter009/9_1_4/index.html

While the example won’t be awarded any esthetics prizes when executed, as you can see in Figure 9.8, it nicely illustrates how the attribute selector has drawn a frame around the `a` and `p` elements because they both contain the `title` attribute.



Attribute selector

Here’s the publisher's website for the book: Rheinwerk Publishing

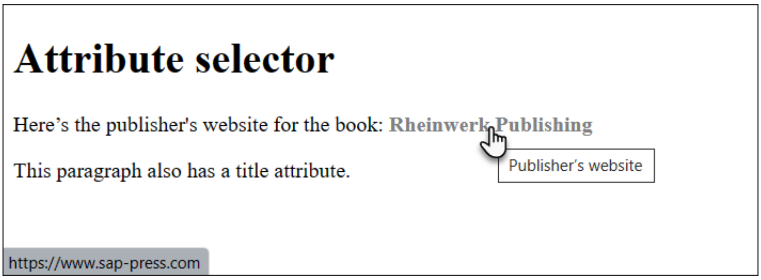
This paragraph also has a title attribute.

Figure 9.8 The Attribute Selector Draws a Frame around the <a> and <p> Elements Because Both Contain the “title” Attribute

Based on this example, you can see that specifying `[attributname]` is the same as specifying the universal selector `*[attributname]`. If you want to address special HTML elements with a specific attribute, you need to connect the attribute selector with the type selector. For example, if you want to style only the `a` elements with the `title` attribute, you can use the following notation:

```
a[title] {
  text-decoration: none;
  color: gray;
  font-weight: bold;
}
```

In Figure 9.9, you can see that, thanks to the union of the type selector and the attribute selector in the HTML document, only the `title` attribute contained in HTML element `<a>` has been selected and styled.


Attribute selector

Here’s the publisher's website for the book: Rheinwerk Publishing

This paragraph also has a title attribute.

Figure 9.9 A Combination of a Type Selector and an Attribute Selector

You can also create such a combination of attribute selectors with class selectors and ID selectors.

Selector	Name	Selection	Example
`[attr]`	Attribute selector (presence)	Any element that contains the `attr` attribute	`<element attr="...">`

Table 9.4 Attribute Selector (Presence of an Attribute)

9.1.5 An Attribute Selector for Attributes with a Specific Value

In addition to selecting attributes, it’s possible to address attributes with a specific value using the attribute selector. Strictly speaking, there are three ways to use an attribute selector to select a particular value:

- The simplest way is to address elements whose attributes contain a certain value. You must write such an attribute selector between square brackets as usual; there you put an equal sign after the attribute name, followed by the attribute value to be selected:

  ```
  [attributename=attributevalue]
  ```

- The second option allows you to address elements whose attributes contain a single word separated by at least one space. To do so, you must write a tilde (~) between the attribute name and the equal sign:

  ```
  [attributename~=attributevalue]
  ```

- The third option can be used to address elements whose attribute value is at the beginning of a string that’s separated by a hyphen (e.g., with a language definition of the `hreflang` or `lang` attribute). Here, you must use the concatenation sign (|) between the attribute name and the equal sign, as follows:

  ```
  [attributename|=attributevalue]
  ```

For demonstration purposes, an HTML document with the attribute selectors presented here gets selected and visually designed. First, the CSS file:

```
/* Styling for all HTML elements where title
   has the attribute value deprecated
 */
[title=deprecated] {
  color: red;
  text-decoration: line-through;
}

/* Styling of HTML elements where title contains the
   the word "Rheinwerk" in the attribute value
 */
[title~=Rheinwerk] { font-weight: bold; }

/* Styling of HTML elements where hreflang
   begins with the attribute value en followed by
   a hyphen
 */
[hreflang|=en] { font-weight: bold; }
```

Listing 9.10 /examples/chapter009/9_1_5/style.css

You can test these attribute selectors using the following HTML document:

```
...
<h1>[title=deprecated]</h1>
<p>The HTML element <code title="deprecated">center</code> has been
   declared deprecated and should be implemented by a CSS solution like
   <code>text-align: center</code>,
   for example.</p>
<h1>[title~=Rheinwerk]</h1>
<ul>
  <li><a href=" https://www.sap-press.com/
         title=" To Rheinwerk-Publishing website">
            Rheinwerk Verlag</a></li>
  <li><a href="http://de.wikipedia.org/wiki/Rheinwerk_Verlag"
         title="To the Wikipedia page of Rheinwerk">
            Rheinwerk at Wikipedia (German language)</a></li>
</ul>
<h1>[hreflang|=en]</h1>
<ul>
  <li><a href="http://en.anywhere.com/" hreflang="en-us">
         US English version</a></li>
  <li><a href="http://en.anywhere.com/" hreflang="de-de">
         UK English version</a></li>
```

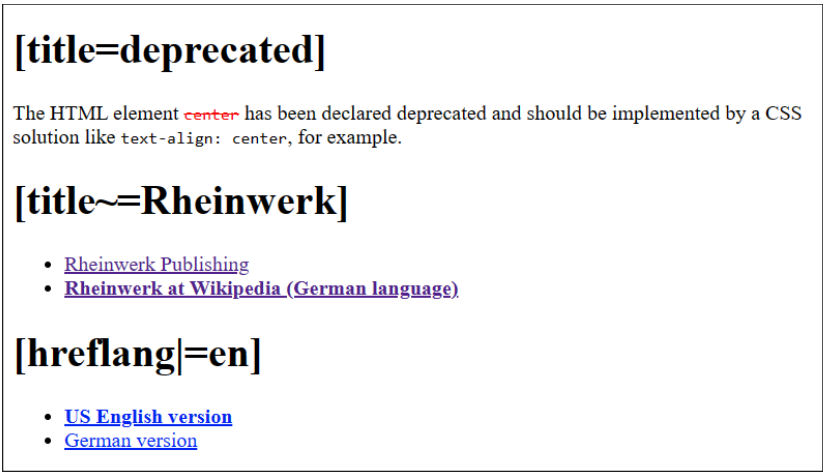
```
</ul>
...
```

Listing 9.11 /examples/chapter009/9_1_5/index.html

The results are displayed in Figure 9.10. The attribute selector `[title=deprecated]` can be used to select any HTML element in the HTML document where the `title` attribute contains the value `deprecated`. In the example, the selected text is marked with red color and crossed out, as it is once included and demonstrated in the HTML document with the HTML element `<code title="deprecated">`.

The second attribute selector `[title~=Rheinwerk]` selects all elements in the document where the `title` attribute contains the word *Rheinwerk*. Here, the word *Rheinwerk* can either be the sole attribute value, which would thus correspond to `[title=Rheinwerk]`, or the word *Rheinwerk* stands alone, separated by spaces. In the example, the first list item was therefore not selected and formatted in bold because there's no space at the end of *Rheinwerk* with `title="To Rheinwerk-Publishing website"`. The attribute value in the second list item, on the other hand, fits the pattern perfectly with `title="To the Wikipedia page of `**`Rheinwerk`**`"`.

In the last example with `[hreflang|=en]`, the first link in the list item is selected because here the pattern matches and the beginning of `hreflang` contains the text string **en** followed by the hyphen, which isn't the case in the second list item. In addition, an element would still be selected if there was only `hreflang="en"`.



[title=deprecated]

The HTML element ~~`center`~~ has been declared deprecated and should be implemented by a CSS solution like `text-align: center`, for example.

[title~=Rheinwerk]

- Rheinwerk Publishing
- **Rheinwerk at Wikipedia (German language)**

[hreflang|=en]

- **US English version**
- German version

Figure 9.10 The Attribute Selectors in Use

Attribute Values with a Digit and Attribute Values without a Value

If the attribute value starts with a digit (e.g., `[title=`**`4u`**`]`), then you must note such special characters between double quotes (e.g., `[title=`**`"4u"`**`]`).

Furthermore, there are attributes that don't need to be assigned a value in HTML, such as the HTML form element for checking off or selecting an option:

```
<input type="checkbox" checked>
```

If you use the `checked` attribute in this form, it will still get an empty string, so you should write the attribute selectors in the following way: `[checked=""]`, `[checked~=""]`, or `[checked|=""]`.

Needless to say, you can also combine these attribute selectors with the type selector to achieve an even more specific selection of attribute values. For example, you can use `a[title=Rheinwerk]` to select only `a` elements where the `title` attribute contains the value `Rheinwerk`.

Selector	Name	Selection	Example
`[attr=value]`	Attribute selector (attribute value)	Any element where the attribute `attr` contains the value `value`	`<elem attr="`**`value`**`">`
`[attr~=value]`	Attribute selector (attribute value)	Any element with the value `value` in the attribute `attr` as a standalone word	`<elem attr="abc `**`value`**` xyz">`
`[attr|=valx]`	Attribute selector (attribute value)	Any element that has the value `valx` in the `attr` attribute at the beginning as a string separated by a hyphen	`<elem attr="`**`valx`**`-valy">`

Table 9.5 Overview of Attribute Selectors (Attribute Value)

9.1.6 Attribute Selector for Attributes with a Specific Partial Value

The attribute selector also provides an extended option to select partial values. The following three attribute selectors are available for this purpose:

- To select an element where the value of the attribute *starts* with a certain string, you only need to note the ^ character between the attribute name and the equal sign:

  ```
  [attributename^=partialvalue]
  ```

- To achieve the counterpart of the attribute selector just mentioned, you can write the dollar sign ($) instead of the circumflex character (^) between the attribute name and the equal sign. Then all values of the attribute ending with a certain string will be selected:

  ```
  [attributename$=partialvalue]
  ```

- To select an element where the value of the attribute is contained as a character string, you need to write the asterisk (*) between the attribute name and the equal sign:

  `[attributename*=partialvalue]`

Again, the presented attribute selectors for selecting partial values in attributes will be briefly demonstrated in use. First, the CSS file:

```
/*
   all a elements where the attribute href starts with http://
 */
a[href^="http://"] {
 text-decoration: none;
 border-bottom: 1px dotted blue; /* dotted blue. Underscore */
}

/*
   all a elements where the href attribute ends with .pdf
 */
a[href$=".pdf"] {
  text-decoration: none;
  color: black;
  padding: 1px;
  border: 2px dotted gray; /* gray frame around it */
}
/*
   a elements where the href attribute contains the text string,
   mydomain
 */
a[href*=mydomain] { font-weight: bold; }  /* Boldface */
```

Listing 9.12 /examples/chapter009/9_1_6/style.css

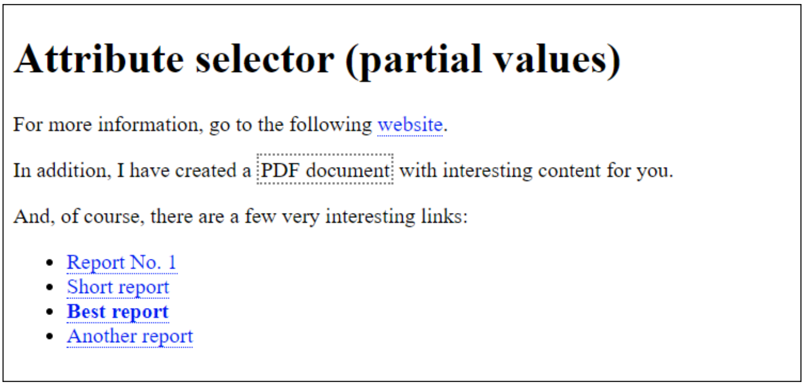
You can test the CSS file *style.css* on the following HTML document:

```
...
<h1>Attribute selector (partial values)</h1>
For more information, go to
   the following <a href="http://domain.com/">website</a>.</p>
<p>In addition, I have created a
   <a href="/documents/document.pdf">
   PDF document</a> with interesting content
   for you.</p>
<p>And, of course, there are a few very interesting links: </p>
  <ul>
```

```
    <li><a href="http://abcdomain.com/report01">
      Report No. 1</a></li>
    <li><a href="http://domainxyz.com/report02">
      Short report</a></li>
    <li><a href="http://mydomain.com/report03">
      Best report</a></li>
    <li><a href="http://vwxdomain.com/report04">
      Another report</a></li>
  </ul>
...
```

Listing 9.13 /examples/chapter009/9_1_6/index.html

You can see the result of this example in Figure 9.11. The attribute selector `a[href^="http://"]` is applied several times to `a` elements in this HTML document where the attribute value of `href` starts with the partial value `http://`, which means it's an external link.



Attribute selector (partial values)

For more information, go to the following website.

In addition, I have created a PDF document with interesting content for you.

And, of course, there are a few very interesting links:

- Report No. 1
- Short report
- **Best report**
- Another report

Figure 9.11 The Extended Attribute Selectors in Use

The second attribute selector `a[href$=".pdf"]` is used only once in the document where, in the `a` element, the attribute value of the `href` attribute ends with `.pdf` (and is thus a PDF document). And the last attribute selector, which contains `a[href*=mydomain]`, is also applied to an `a` element where the attribute value of `href` contains the text string `mydomain`; this is styled with boldface.

Selector	Name	Selection	Example
`[attr^=value]`	Attribute selector (partial value)	Any element where the attribute `attr` starts with the text string `value`	<elem attr="**value**less">

Table 9.6 Attribute Selectors (Partial Value)

Selector	Name	Selection	Example
`[attr$=value]`	Attribute selector (partial value)	Any element where the attribute `attr` ends with the text string `value`	`<elem attr="outvalue">`
`[attr*=value]`	Attribute selector (partial value)	Any element where the attribute `attr` contains the text string `value`	`<elem attr="misvalued">`

Table 9.6 Attribute Selectors (Partial Value) (Cont.)

9.1.7 CSS Pseudo-Classes: The Selectors for Specific Features

In HTML, there are elements you can't access using ordinary selectors. These include, for example, elements over which the mouse pointer is currently located, or a hyperlink that has already been visited or not. Many of those properties can be addressed via pseudo-classes, which will be briefly described in the following sections.

Pseudo-Classes for Visited and Nonvisited Hyperlinks

The pseudo-classes `:link` and `:visited` allow you to select and specially mark unvisited and visited links, respectively. A simple example might look as follows:

```
a:link { color: red; }
a:visited { color: green; }
...
```

Listing 9.14 /examples/chapter009/9_1_7/css/style.css

In this example, all `a` elements whose reference hasn't been visited yet are displayed in red, while the references that have already been visited are displayed in green. You can find an example here: */examples/chapter009/9_1_7/index.html*.

Selector	Name	Selection	Example
`:link`	Link pseudo-class	An unvisited hyperlink	`a:link{ color: blue; }`
`:visited`	Link pseudo-class	A visited hyperlink	`a:visited{ color: gray; }`

Table 9.7 Link Pseudo-Classes for Visited and Nonvisited Links

There's also a newer pseudo-class, `:any-link`, which selects all links, whether visited or not. This way, you can select all elements that contain an `href` attribute, for example:

```
...
.articlestyle a:any-link { color: gray; }
```

Listing 9.15 /examples/chapter009/9_1_7/css/style.css

This will display all `a` elements contained in the `articlestyle` class in gray.

Pseudo-Classes for User Interactions with Mouse and Keyboard

You can use the CSS pseudo-classes `:hover`, `:active`, and `:focus` to respond to different user interactions. Using `:hover`, you can select elements over which the mouse pointer is located. The pseudo-class `:focus` selects the elements that receive focus (e.g., via the `Tab` key), while `:active` selects elements that are currently clicked. What's also interesting in connection with the `input` element is the pseudo-class `:placeholder-shown`, which enables you to address the `placeholder` text of an `input` element.

Here's a simple example that demonstrates all four pseudo-classes in use:

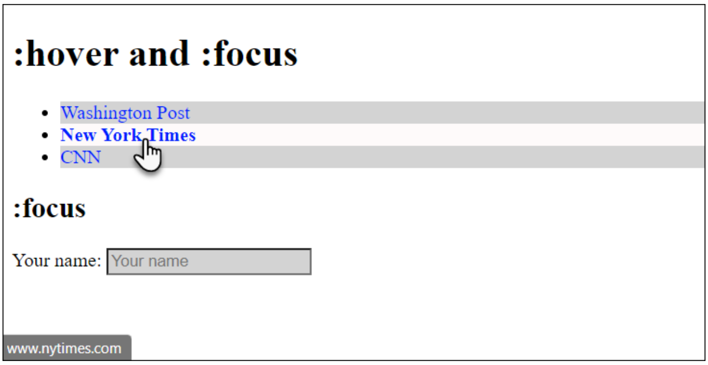
```
input { background-color: lightgray; }
input:focus { background-color: white; }
input:hover { box-shadow: 0 0 3px blue;  }
input:placeholder-shown { color: white; }
li { background-color: lightgray; }
li:hover { background-color: snow; }
li:active{ background-color: gray; }
a:link { text-decoration: none; color: blue; }
a:hover { font-weight: bold; }
a:active { color: red; }
```

Listing 9.16 /examples/chapter009/9_1_7/css/style2.css

```
...
    <h1>:hover and :focus</h1>
    <ul>
      <li><a href="http://www.washingtonpost.com/">Washington Post</a></li>
      <li><a href="http://www.nytimes.com/">New York Times</a></li>
      <li><a href="http://www.cnn.com/">CNN</a></li>
    </ul>
    <h2>:focus</h2>
    <form>
    Your name:
      <input type="text" name="name" id="name" placeholder="Your name" />
    </form>
...
```

Listing 9.17 /examples/chapter009/9_1_7/index2.html

Figure 9.12 shows the result of the HTML document with the pseudo-classes in use. Here, for example, `input:focus` was used to cause the input field containing the HTML element `<input>` to be displayed with a white background when the text input field receives focus. With `input:hover`, on the other hand, a blue frame is displayed around the input field when you pause over it with the mouse cursor. `input:placeholder-shown`, in turn, was used to display the `placeholder` text in white color.



:hover and :focus

- Washington Post
- New York Times
- CNN

:focus

Your name: Your name

www.nytimes.com

Figure 9.12 The Pseudo-Classes for an Interactive User Input in Use: Pseudo-Class ":hover"

Pseudo-Classes for Form Elements

The styling of HTML forms will be described in Chapter 14, Section 14.7, but it should be mentioned here that there are also three dynamic pseudo-classes available for the form elements: `:enabled` (selectable; enabled), `:disabled` (not selectable; disabled), and `:checked` (checked). Strictly speaking, these are *user interface pseudo-classes*.

You can use `li:hover` or `a:hover` to respond when a user hovers over the HTML element `<li>` or `<a>` by changing the background color or font style to bold. With `li:active` and `a:active`, you respond when a user activates or left-clicks the HTML elements `<li>` and `<a>`, respectively.

It's important to maintain the order of the pseudo-class selectors, that is `:link`, `:visited`, `:hover`, `:focus`, and `:active` (LVHFA); otherwise, `:visited` will overwrite the `:hover` pseudo-class. If you ever have problems with a dynamic pseudo-class not working as intended, it's probably because one pseudo-class has overridden another. In that case, you should check the order of the pseudo-classes. To help you remember the LVHFA order, you can use a mnemonic such as *Lord Vader Hates Furry Animals*.

":hover" and Tablets or Smartphones

While the pseudo-classes can now be used for any HTML element, you should still keep in mind that there's no `:hover` available on smartphones or tablets—that is, that a mouse is placed over an element.

Selector	Name	Selection	Example
`:hover`	Dynamic pseudo-class	The element is being hovered over by the user moving the mouse cursor over a hyperlink or element.	`a:hover { color: red; }`
`:focus`	Dynamic pseudo-class	The element has the focus (e.g., input field with active cursor).	`input:focus{ background-color: yellow; }`
`:active`	Dynamic pseudo-class	The element is being activated (e.g., clicked on with the mouse).	`a:active { color: green; }`
`:placeholder-shown`	Dynamic pseudo-class	The element is used to address the `placeholder` text of an `input` element.	`input:placeholder-shown { color: white; }`

Table 9.8 Dynamic Pseudo-Classes for Mouse and Keyboard Interaction

The ":target" Pseudo-Class for Reference Targets

Another pseudo-class is `:target`, which can be applied to elements that are the target of a reference. The CSS rule that's written with it doesn't become active until the reference target has been jumped to. To activate the reference target, you can use an ID in addition to an ordinary anchor (see Chapter 5, Section 5.2.7).

A simple example shows the `:target` pseudo-class in use:

```
:target { background: lightgray; }
div#show { display: none; }
div#show:target { display: block; }
```

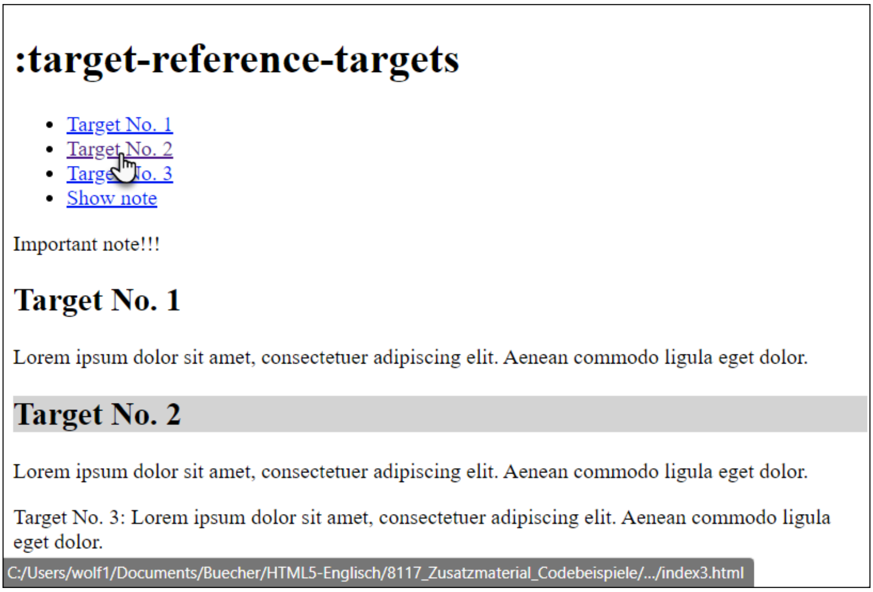
Listing 9.18 /examples/chapter009/9_1_7/css/style3.css

```
...
<h1>:target-reference-targets</h1>
  <ul>
    <li><a href="#target01">Target No. 1</a></li>
    <li><a href="#target02">Target No. 2</a></li>
    <li><a href="#target03">Target No. 3</a></li>
    <li><a href="#show">Show note</a></li>
  </ul>
<div id="show">Important note!!!</div>
<h2 id="target01">Target No. 1</h2>
<p>Lorem ipsum dolor sit amet, consectetuer adipiscing ... </p>
```

```
<h2><a name="target02">Target No. 2</a></h2>
<p>Lorem ipsum dolor sit amet, consectetuer adipiscing ... </p>
<p id="target03">Target No. 3: Lorem ipsum dolor sit amet ... </p>
...
```

Listing 9.19 /examples/chapter009/9_1_7/index3.html

Here, the pseudo element `:target` was used to display the reference target that's currently being jumped to in gray background color. In the example, you've defined three reference targets: `target01`, `target02` and `target03`. The reference targets `target01` and `target02` are the `h2` elements, while `target03` is a `p` element that's also grayed out when you select the reference target. Another possible use of `:target` can be found with `div#show:target`, where you first used `div#show` and `display: none;` to hide the `div` element with the `show` ID when loading the page. If the visitor selects the reference target **Show note**, the text between `<div>` and `</div>` will show with the `Show` ID and use `display: block;`. You can use the CSS feature `display` to specify how an element should be displayed or change the behavior of an element. The use of `display: block;` and `display: none;` to show and hide HTML elements is a commonly cited example.



:target-reference-targets

- Target No. 1
- Target No. 2
- Target No. 3
- Show note

Important note!!!

Target No. 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

Target No. 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

Target No. 3: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/.../index3.html

Figure 9.13 The Pseudo-Class ":target" for "Target No. 2", Resulting in the Heading Now Being Displayed with a Gray Background

9.1.8 The Convenient Structural Pseudo-Classes in CSS

The structural pseudo-classes allow you to select elements based on their position in the document structure.

Addressing the Root of a Document via ":root" and Empty Elements via ":empty"

The two structural pseudo-classes **`:root`** and **`:empty`** can be used to select either the root of the document (`:root`) or empty elements (`:empty`). Here's a simple example that demonstrates these two pseudo-classes in use:

```
:root { color: lightgray; }
:empty { background-color: yellow; padding: 10px; }
td:empty { background-color: green; }
```

Listing 9.20 /examples/chapter009/9_1_8/css/style.css

```
<!doctype html>
<html>
  <head>
...
  <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
    <h1>:root and :empty</h1>
    <p>Lorem ipsum dolor sit amet, consectetuer ...</p>
    <p></p>
    <h2>:empty on table</h2>
    <table>
      <tbody>
        <tr><td>Value</td><td></td></tr>
        <tr><td></td><td>Value</td></tr>
       <tr><td></td><td></td></tr>
      </tbody>
    </table>
  </body>
</html>
```

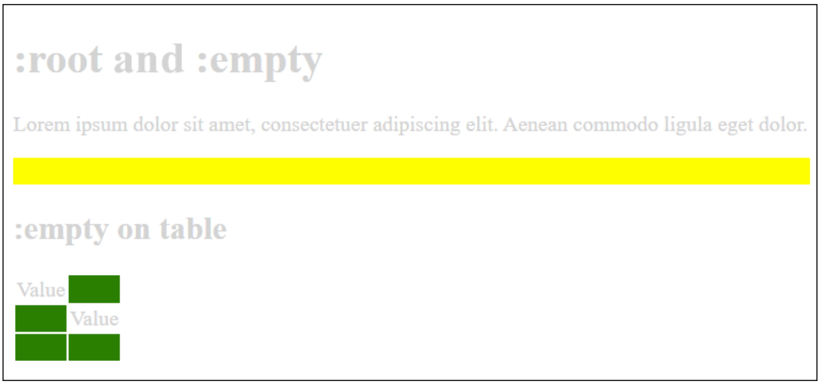
Listing 9.21 /examples/chapter009/9_1_8/index.html

Compared to Figure 9.14, you can see that `:root`, for which a gray text color was used, affects the entire document. Thus, the use of the structural pseudo-class `:root{}` corresponds to that of the type selector `html{}`, the only difference being that it has a higher weighting. If your web browser doesn't know `:root`, the text won't be formatted in gray.

The structural pseudo-class `:empty`, on the other hand, has an effect on empty elements such as the empty paragraph `<p></p>`, which has been formatted in yellow. More specifically, in conjunction with a type selector, `td:empty`, the empty table cells have been styled with green color.

However, `:empty` is limited to elements that contain nothing at all or only one comment. If a whitespace such as a space, tab feed, or line break gets included in the element,

`:empty` will no longer access it. For this purpose, you can use the structural pseudo-class `:blank`. `:blank` works like `:empty`, except that it also selects empty elements with whitespace characters. However, this structural pseudo-class isn't yet natively supported by any web browser. Only Firefox allows you to test this pseudo-class by using the vendor prefix `:-moz-only-whitespace`.



:root and :empty

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

:empty on table

Value	🟩
🟩	Value
🟩	🟩

Figure 9.14 Effects of the Pseudo-Classes ":root" and ":empty" on the HTML Document

Structural Pseudo-Classes for Child Elements

The structural pseudo-classes `:first-child`, `:last-child`, `:nth-child()`, `:nth-last-child()`, and `:only-child` allow you to select specific child elements in the HTML document structure. A child element (or descendant element) is an element below a given element when you look at the Document Object Model (DOM) tree.

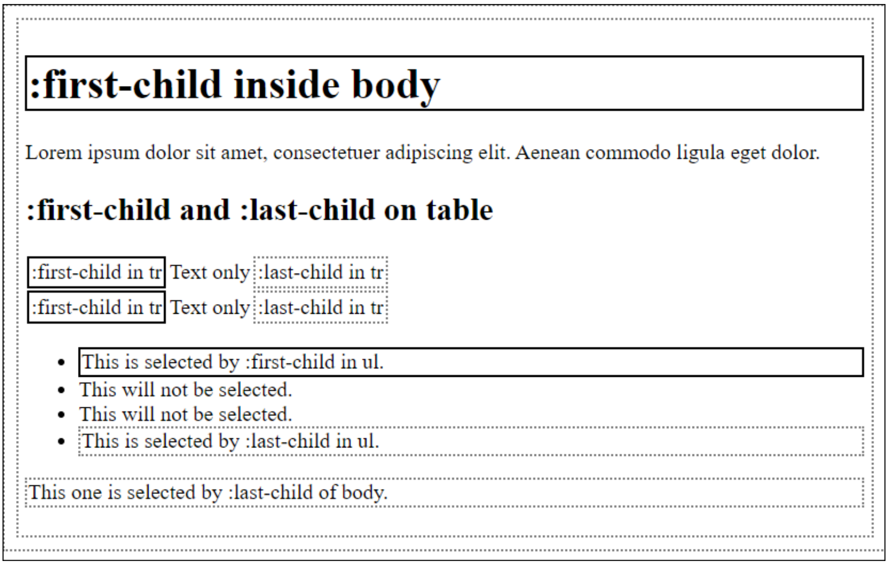
You can use the structural pseudo-class `:first-child` to select the first child element of an HTML element. The counterpart to this, `:last-child`, can be used to select the last child of a parent element in the document structure.

Figure 9.15 shows an example with the pseudo-class selectors `:first-child` and `:last-child` in use. The CSS code for this can be found in the ZIP file located in */examples/chapter009/9_1_8/css/style2.css* and the corresponding HTML document in */examples/chapter009/9_1_8/index2.html*. The selected `:first-child` elements were framed with a solid black line for clarity, while `:last-child` elements were framed with a dotted gray line. The example shows that `body` is also a `:last-child` element of the `html` element. The first element of `html` is `head`.

The structural pseudo-class `:nth-child()`, on the other hand, allows you to select the nth child element of a parent element. As an argument, this selector expects an integer value or an arithmetic calculation that returns an integer value. Possible arguments you can use are the values `odd` (odd numbers) and `even` (even numbers). The structural pseudo-class `:nth-last-child()` works in a similar way, except that here the counting starts at the end. These pseudo-classes can be used, for example, to alternately color table rows, which greatly simplifies the readability of long tables:

```
...
tr:nth-child(odd) { background: lightgray; }
tr:nth-child(even) { background: gray; }
...
```

Listing 9.22 /examples/chapter009/9_1_8/css/style3.css



:first-child inside body

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

:first-child and :last-child on table

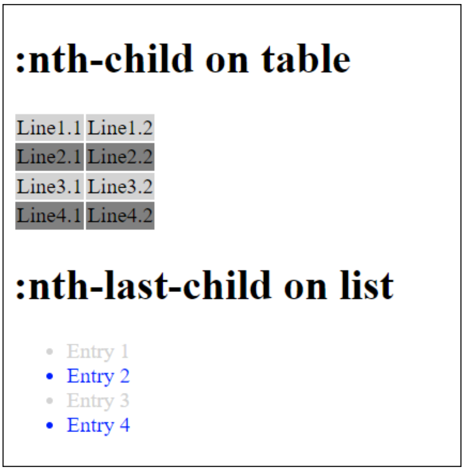
:first-child in tr	Text only	:last-child in tr
:first-child in tr	Text only	:last-child in tr

- This is selected by :first-child in ul.
- This will not be selected.
- This will not be selected.
- This is selected by :last-child in ul.

This one is selected by :last-child of body.

Figure 9.15 The Pseudo-Class Selectors ":first-child" and ":last-child" in Use

As you can see in Figure 9.16, each odd table row (`tr` element) is styled with `lightgray`, and each even table row is styled with `gray` color.



:nth-child on table

Line1.1	Line1.2
Line2.1	Line2.2
Line3.1	Line3.2
Line4.1	Line4.2

:nth-last-child on list

- Entry 1
- Entry 2
- Entry 3
- Entry 4

Figure 9.16 The Pseudo-Classes ":nth-child()" and ":nth-last-child()" in Use (HTML Document Available in /examples/chapter009/9_1_8/index3.html)

Another structural pseudo-class is `:only-child`, which is used to select only those elements in the HTML document which are the only child element of a parent element. For example, for a list with HTML element `<ul>`, this would be the case if there was only one `li` element in the list.

Structural Pseudo-Classes for Specific Child Elements

The structural selectors `:first-of-type`, `:last-of-type`, `:nth-of-type()`, `:nth-last-of-type()`, and `:only-of-type` apply only to specific child elements. This is in contrast to the `:...child` selectors, which can apply to all child elements.

Accordingly, you can use `:first-of-type` or `:last-of-type` to select the first or last child element of a particular HTML element. Here's a simple example you couldn't have implemented with `:first-child` or `:last-child`:

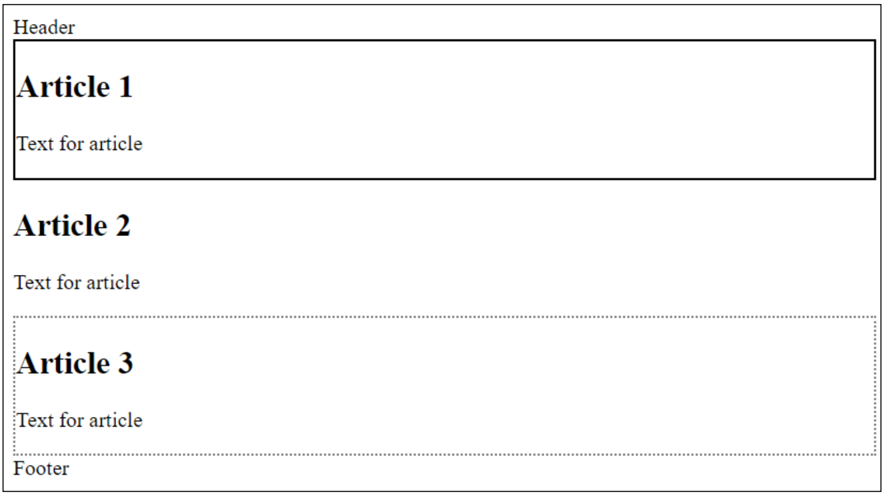
```
article:first-of-type { border: 2px solid black; }
article:last-of-type { border: 2px dotted gray; }
```

Listing 9.23 /examples/chapter009/9_1_8/css/style4.css

```
    ...
    <link rel="stylesheet" href="css/style4.css">
  </head>
  <body>
      <header>Header</header>
      <article>
        <h1>Article 1</h1>
        <p>Text for article</p>
      </article>
      <article>
        <h1>Article 2</h1>
        <p>Text for article</p>
      </article>
      <article>
        <h1>Article 3</h1>
        <p>Text for article</p>
      </article>
      <footer>Footer</footer>
  </body>
</html>
```

Listing 9.24 /examples/chapter009/9_1_8/index4.html

In this example, `article:first-of-type` and `article:last-of-type` select the first and last `article` elements respectively. Because of the `header` and `footer` elements, in this case you can't select the first and last `article` elements with `:first-child` and `:last-child`, respectively, and must use `:first-of-type` and `:last-of-type` instead.



Header

Article 1

Text for article

Article 2

Text for article

Article 3

Text for article

Footer

Figure 9.17 The Pseudo-Class Selectors ":first-of-type" and ":last-of-type" in Use

If you had wanted to select the second article in the example in Figure 9.17, you would have done this with the following:

```
article:nth-of-type(2) {...}
```

Again, you would have to use the `:nth-of-type()` version instead of the `:nth-child()` version because you would have used `article:nth-child(2)` to select the first `article` element; the `header` element would have been the first child in this example, and the first `article` element would have been the second `child`. The counterpart to `:nth-of-type()` is `:nth-last-of-type()`, except that the counting of elements starts at the end.

Finally, the `of-type` counterpart to `:only-child`, that is, `:only-of-type`, allows you to address an element if it's the only element of the type in the parent element. Let's look at a simple example:

```
em:only-of-type { font-weight: bold; }
```

Listing 9.25 /examples/chapter009/9_1_8/css/style5.css

```
...
  <link rel="stylesheet" href="css/style5.css">
...
```

```
<body>
  <h1>:only-of-type</h1>
  <p><em>Bear</em>! Who is this <em>Bear</em>?</p>
  <p>Caution! <em>Bear</em> could be standing <strong>behind</strong>.
     you!</p>
</body>
...
```

Listing 9.26 /examples/chapter009/9_1_8/index5.html

In this example, only the `em` element in the second paragraph is selected with `em:only-of-type` because it's the *only* `em` element in the `p` element. The `strong` element isn't important in such a case. With the `em:only-child` version, no `em` element would have been selected at all because there's no `only` element in the two `p` elements.

Finally, here's a table with an overview of the various structural pseudo-classes that are available.

Selector	Selects
`:root`	Root element.
`:empty`	Empty elements.
`:first-child`	An element that is the first of its parent element.
`:last-child`	An element that is the last of its parent element.
`:nth-child(n)`	An element that is the `n`th child element. For `n`, the values `odd` (odd numbers) and `even` (even numbers) are also possible.
`:nth-last-child(n)`	Like `:nth-child(n)`, except that here the count starts at the end.
`:only-child`	An element that is the only child element in the parent element.
`:first-of-type`	An element that is the first child element of a given type.
`:last-of-type`	An element that is the last child element of a given type.
`:nth-of-type(n)`	An element that is the `n`th identical child element of a parent element. For `n`, the values `odd` (odd numbers) and `even` (even numbers) are also possible.
`:nth-last-of-type(n)`	Like `:nth-of-type(n)`, except that the counting starts at the end.
`:only-of-type`	An element if it's the only child of this type in the parent element.

Table 9.9 Overview of the Structural Pseudo-Classes

9.1.9 Other Useful Pseudo-Classes

CSS provides some more pseudo-classes, which I'll briefly describe here. First, there's the language pseudo-class `:lang()`, which you can use to select elements you've provided with the `lang` attribute. The corresponding element gets selected together with its descendants. For example, if you want to style all elements written with `lang="en"` in blue text color, you can do it as follows:

```
:lang(en) { color: blue; }
```

Another very useful pseudo-class is the negation pseudo-class `:not()`, which allows you to select elements with which a selector does *not* match. This pseudo-class expects a simple selector as an argument. With the following simple example, you can make sure that all elements which aren't text paragraphs (i.e., not `p` elements) get a gray text color:

```
:not(p) { color: gray; }
```

All this can be specified in more detail as follows:

```
#content:not(p) { color: gray; }
```

Now all child elements of the element with ID `content` that aren't `p` elements get a gray color. Such combinations can be expanded even further, for example:

```
article:not(.news) { ... }
```

This selects all `article` elements whose elements don't contain the `news` class.

In the future, it will also be possible to use more than one selector as an argument, for example:

```
p:not(.advertisment, .news) { color: gray; }
```

This way, you can style all `p` elements in gray font color. Elements that are provided with the `advertisement` or `news` class are excluded from this. At the time of printing this book, however, only the Safari web browser was able to handle multiple selectors.

The counterpart of the pseudo-class `:not()` is `:matches()`. This allows you to select multiple elements that match the selector, for example:

```
p:matches(.advertisement, .news) { color: gray; }
```

This will gray out all `p` elements that contain the advertisement or news class. When this book was printed, Safari could already use this natively. In Chrome and Edge, you still need to use `:-webkit-any()` for it.

Selector	Name	Description
`:lang(xx)`	Language pseudo-class	Selects elements where the language has been provided with the `lang="xx"` attribute
`:not(s)`	Negation pseudo-class	Selects elements to which selector `s` doesn't apply
`:matches(s1,s2, ...)`	Matches pseudo-class	Selects all elements to which selectors `s1` and `s2 ...` apply

Table 9.10 Language Pseudo-Class and Negation Pseudo-Class

9.1.10 Pseudo-Elements: The Selectors for Nonexistent Elements

CSS pseudo-elements provide you with another group of selectors. This allows you to address elements that aren't directly present as elements in the structure of the HTML document, but are still resulting from the structure with HTML or are generated using the CSS feature `content`.

Like pseudo-classes, pseudo-elements in CSS start with a colon (`:pseudo-element`). However, they should be used with two colons (`::pseudo-element`) to better distinguish them from CSS pseudo-classes.

The Pseudo Elements "::first-letter" and "::first-line"

You can use `::first-letter` to select and style the first character in an HTML element. And as you might guess from the name, the CSS pseudo-element `::first-line` selects the first line in an HTML element.

The Pseudo-Elements "::before" and "::after"

The CSS pseudo-elements `::before` and `::after` allow you to add content to an existing content. These pseudo-elements are often used in conjunction with the CSS feature `content` to add content before (`::before`) or after (`::after`) the existing content.

The Pseudo-Element "::selection"

The CSS pseudo-element `::selection` selects the text marked by the user. This can be useful if the color scheme of the layout doesn't match the one used when selecting text, which is usually white on blue. However, only some CSS features can be used with this pseudo-element. These include `color`, `background` (`background-color`, `background-image`), and `text-shadow`.

Here's a simple example you can use to test some of the pseudo-elements presented here:

```
p::first-line { font-weight: bold; }
p::first-letter{ font-size: xx-large; float: left; }
td.time::before { content: "ca. "; }
td.time::after  { content: " minutes"; }
```

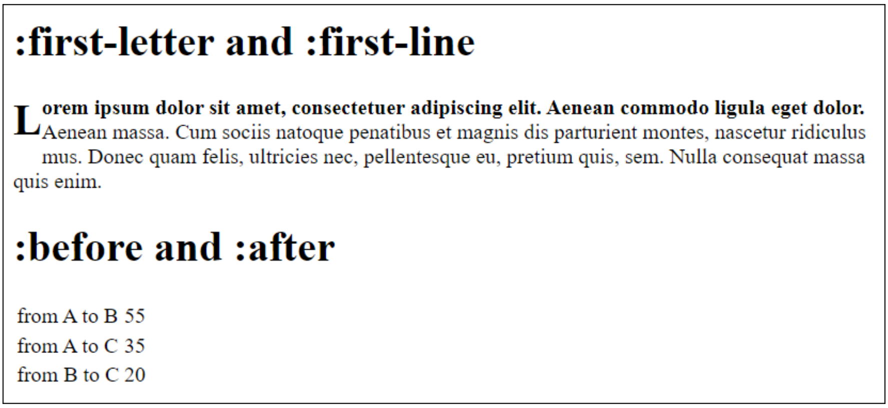
Listing 9.27 /examples/chapter009/9_1_10/css/style.css

```
...
    <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
     <h1>:first-letter and :first-line</h1>
     <p>Lorem ipsum dolor sit amet, consectetuer adipiscing.
        Aenean commodo ligula eget dolor. Aenean massa. ...</p>
     <h1>:before and :after</h1>
     <table>
      <tbody>
        <tr><td>from A to B</td><td class="time">55</td></tr>
        <tr><td>from A to C</td><td class="time">35</td></tr>
        <tr><td>from B to C</td><td class="time">20</td></tr>
      </tbody>
    </table>
  </body>
...
```

Listing 9.28 /examples/chapter009/9_1_10/index.html

You can use `p::first-line` to select the first line of the `p` element that can be displayed in the web browser and output it in bold. `p::first-letter`, on the other hand, allows you to select the first letter in the `p` element and highlight it clearly with a larger font. Used in this way, this applies to all `p` elements in the HTML document. On the other hand, if you have a `p` element such as `<p class="abc">`, you can also access it via `p.abc::first-line`, which is a composite of type selector, class selector, and pseudo-element.

`td.time::before` enables you to insert the text `"approx. "` before the content of a table cell that contains the `time` class by means of the CSS feature `content`. In a similar way, you can use `td.time::after`, except that here the content `" minutes"` is inserted at the end of the content of the table cells. Figure 9.18 shows the example in use.



:first-letter and :first-line

L**orem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.** Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

:before and :after

from A to B 55
from A to C 35
from B to C 20

Figure 9.18 The Pseudo-Elements in Use

Selector	Description
`::first-letter`	Selects the first character in a line
`::first-line`	Selects the first line in a paragraph
`::before`	Inserts content before an element and formats it
`::after`	Inserts content after an element and formats it

Table 9.11 Overview of the Pseudo-Elements

9.2 Combinators: Concatenating the Selectors

A *combinator* is a character between two selectors that concatenates these selectors. Here, the first selector represents the condition, while the second selector forms the target to be selected if the condition is true. In CSS, you have four such combinators, which in turn you can concatenate with other combinators.

Combinator	Name	Meaning
`E F`	Descendant selector (*descendant combinator*)	`F` gets selected if it's a descendant of an `E` element.
`E > F`	Child selector (*child combinator*)	`F` gets selected only if it's a direct descendant of an `E` element.
`E + F`	Adjacent sibling selector (*adjacent sibling combinator*)	`F` gets selected only if it occurs directly after `E` (in the same parent element).
`E ~ F`	General sibling selector (*general sibling combinator*)	`F` gets selected only if it occurs after `E` (in the same parent element).

Table 9.12 Quick Overview of the Different Combinators

For illustration purposes, all four combinators are used in a simple way in the following example. In practice, of course, such combinators can become far more complex in the way they are used. In our example, simple type selectors are combined, but you can also combine class or ID selectors, for example. Inside the negation pseudo-class `:not()`, you can't use any combinators.

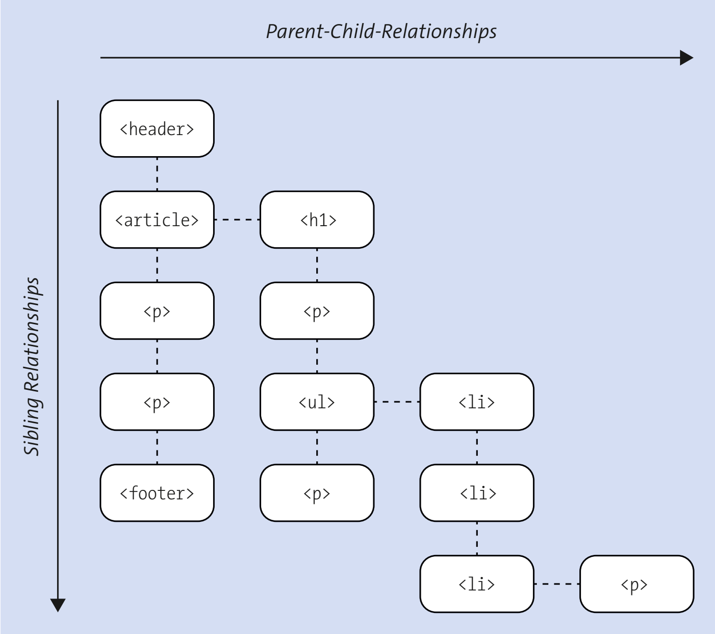


Figure 9.19 Document Structure Tree of the Example

```
...
  <body>
    <header>Header</header>
    <article>
        <h1>Article 1</h1>
        <p>1. Paragraph text for article</p>
        <ul>
          <li>List item 1</li>
          <li>List item 2</li>
          <li>
            <p>A paragraph text in the list item</p>
          </li>
        </ul>
        <p>2. Paragraph text for article</p>
    </article>
    <p>1. Paragraph text after the article</p>
    <p>2. Paragraph text after the article</p>
    <footer>Footer</footer>
  </body>
...
```

Listing 9.29 /examples/chapter009/9_2/index.html

The goal in this example is to access the individual `p` elements. For this purpose, the `article` element should be the condition and the `p` element the target. For a better understanding, the document structure tree is shown in Figure 9.19, which demonstrates this source code a bit more clearly in the hierarchy of the individual HTML elements.

9.2.1 The Descendant Combinator (E1 E2)

The *descendant combinator* is the oldest of all combinators and connects two selectors with one space. It allows you to select all children and children's children of an element. With regard to the */examples/chapter009/9_2/index.html* example, the descendant combinator should be used as follows:

```
article p { background: lightblue; }
```

Listing 9.30 /examples/chapter009/9_2_1/css/style.css

This code listing specifies that all `p` elements located within an `article` element will be selected and assigned a light blue background color. Figure 9.20 shows this process in the document structure tree, and Figure 9.21 shows the example in actual use.

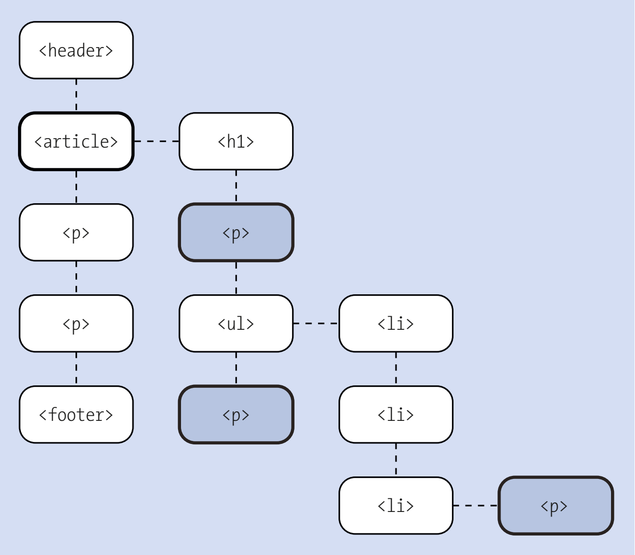
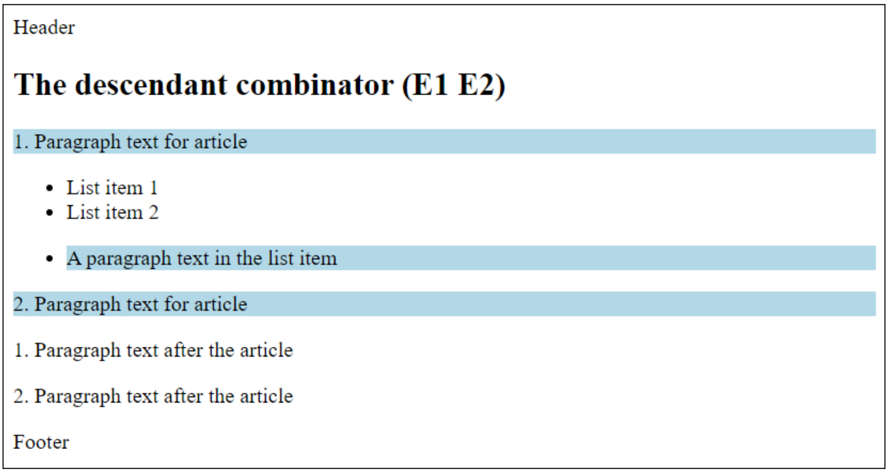


Figure 9.20 You Can Use the Descendant Combinator to Select All Child and Children's Children Elements That Were Specified as the Target (i.e., <p> Element)



Header

The descendant combinator (E1 E2)

1. Paragraph text for article

- List item 1
- List item 2

- A paragraph text in the list item

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 9.21 The Example of the Descendant Combinator in use

Combinator	Name	Meaning
`E F`	Descendant selector (*descendant combinator*)	`F` gets selected if it's a descendant of an `E` element.

Table 9.13 The Descendant Combinator (E1 E2)

Warning: A Common Mistake!

A common error occurs when you separate multiple selectors with or without commas. Let's look at an intentional notation such as the following:

```
h2, p {...}
```

This addresses all `h2` and `p` elements in an HTML document. If the comma gets omitted by mistake such as the following:

```
h2 p {...}
```

Then, this expression receives a different meaning with the space combinator, and theoretically only those `p` elements would be selected that are located inside `<h2>` and `</h2>`, which doesn't make any sense in that case. Although for this purpose, the W3C suggests for selectors (level 4) to use the `>>` character for the descendant combinator (i.e., `E >> F` corresponds to `E F`), this hasn't been implemented by any browser vendor up to now.

9.2.2 The Child Combinator (E1 > E2)

The child *combinator* is represented by a closing angle bracket (>) and connects two selectors. By connecting to the child combinator, you select only elements that are direct descendants of the parent element. Children's children are no longer selected here. With regard to the */examples/chapter009/9_2/index.html* example, the child combinator can be demonstrated as follows:

```css
article > p { background: lightblue; }
```

Listing 9.31 /examples/chapter009/9_2_2/css/style.css

This way, you can select all `p` elements that are direct descendants of the `article` element and style them with a light blue background. Figure 9.22 shows this process in the document structure tree, while Figure 9.23 shows the example in actual use.

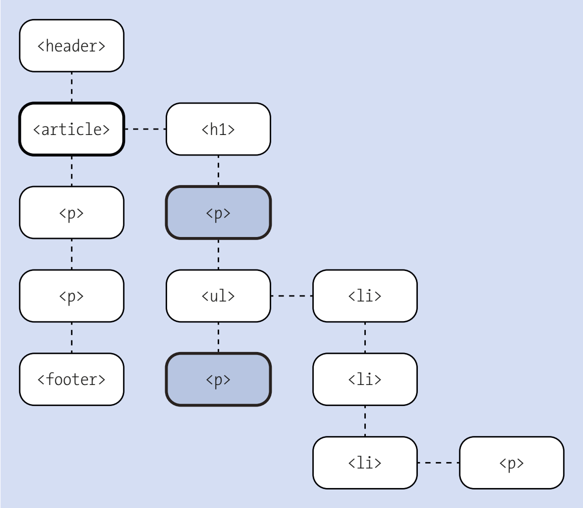
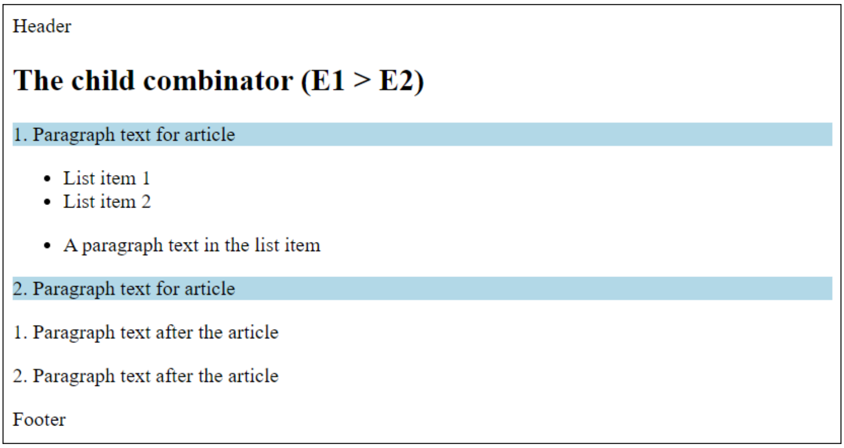


Figure 9.22 You Can Use the Child Combinator (with "article > p {...}") to Select Only the Direct Child Elements



Header

The child combinator (E1 > E2)

1. Paragraph text for article

- List item 1
- List item 2

- A paragraph text in the list item

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 9.23 The Example of the Child Combinator in Use: Only the Direct Child Elements Are Selected

Combinator	Name	Meaning
`E > F`	Child selector (*child combinator*)	`F` gets selected only if it's a direct descendant of an `E` element.

Table 9.14 The Child Combinator (E1 > E2)

9.2.3 The Adjacent Sibling Combinator (E1 + E2)

In the adjacent sibling combinator, sometimes called a direct sibling selector, two selectors can be connected with each other by means of the plus sign (+). This way, you can only address elements that are immediate neighbors on the same level (i.e., they have the same parent element). Once again, we can use the */examples/chapter009/9_2/index.html* example to illustrate the adjacent sibling combinator:

```
article + p { background: lightblue; }
```

Listing 9.32 /examples/chapter009/9_2_3/css/style.css

Here, only the `p` element is selected, which is a descendant of the `article` element. The `article` and `p` elements have the same parent element (the `body` element in the example). Figure 9.24 shows this process in the document structure tree, and Figure 9.25 shows the example in actual use.

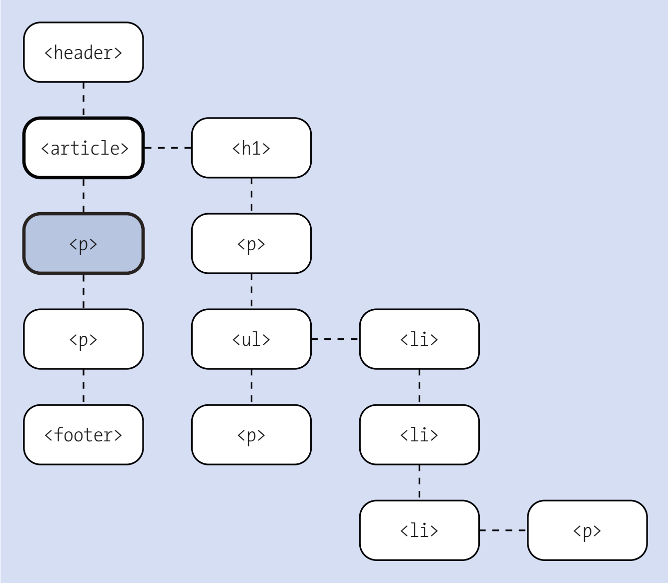
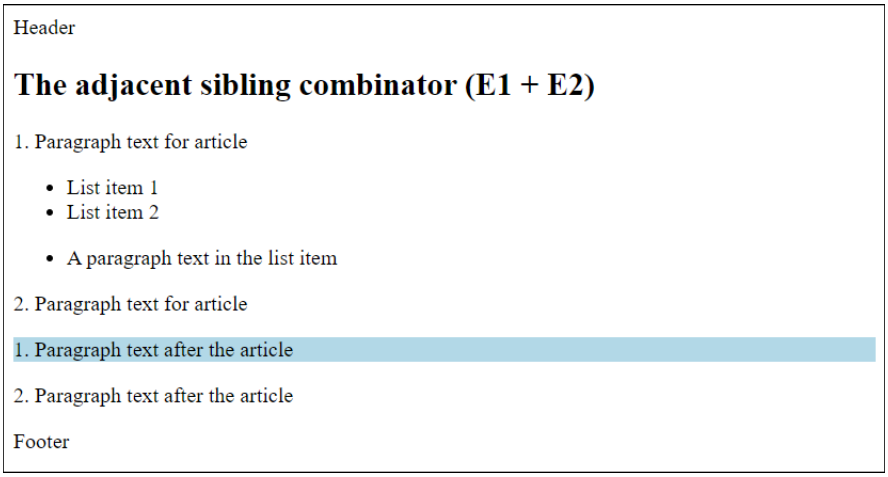


Figure 9.24 With the Adjacent Sibling Combinator, Only Elements That Are Immediate Neighbors on the Same Level (i.e., Have the Same Parent Element) Are Selected

Combinator	Name	Meaning
E + F	Adjacent sibling selector (*adjacent sibling combinator*)	F gets selected only if it occurs directly after E (in the same parent element).

Table 9.15 The Adjacent Sibling Combinator



Header

The adjacent sibling combinator (E1 + E2)

1. Paragraph text for article

- List item 1
- List item 2

- A paragraph text in the list item

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 9.25 The Example of the Adjacent Sibling Combinator in Use

9.2.4 The General Sibling Combinator (E1 ~ E2)

The (indirect) general sibling combinator allows you to connect two selectors by means of the tilde character (~). This means it also allows you to address elements that are neighbors on the same level (= have the same parent element). In contrast to the adjacent sibling combinator (+) (or direct sibling selector), these elements don't need to follow each other directly, and other elements can also be located between them. This sounds more complex than it really is, which is why the document */examples/chapter009/9_2/index.html* is used again here for illustration purposes:

```
article ~ p { background: lightblue; }
```

Listing 9.33 /examples/chapter009/9_2_4/css/style.css

This selects all `p` elements that follow an `article` element and are at the same level in the document tree. Figure 9.26 shows this process in the document structure tree, while Figure 9.27 shows the example in actual use. The general sibling combinator selects all adjacent `<p>` elements after the `<article>` element that are on the same level. It doesn't matter if there are other elements like `<p>` in between.

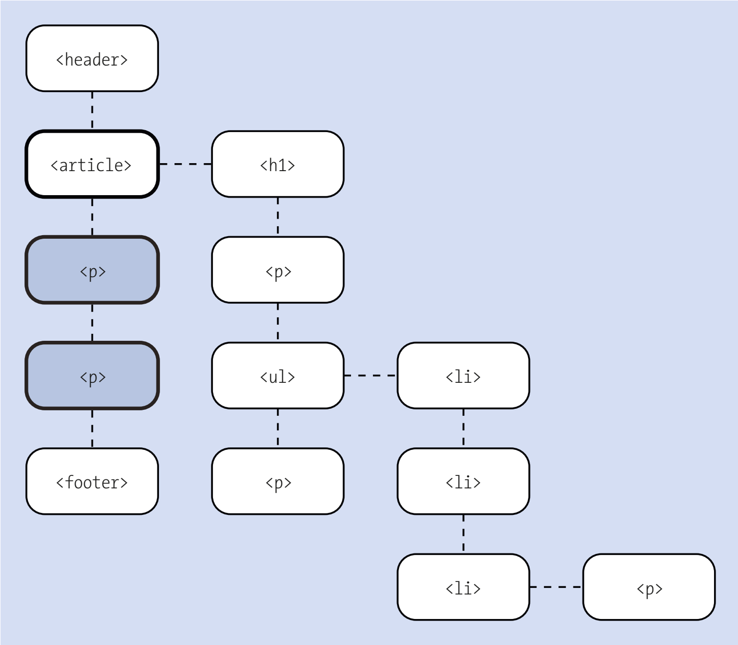
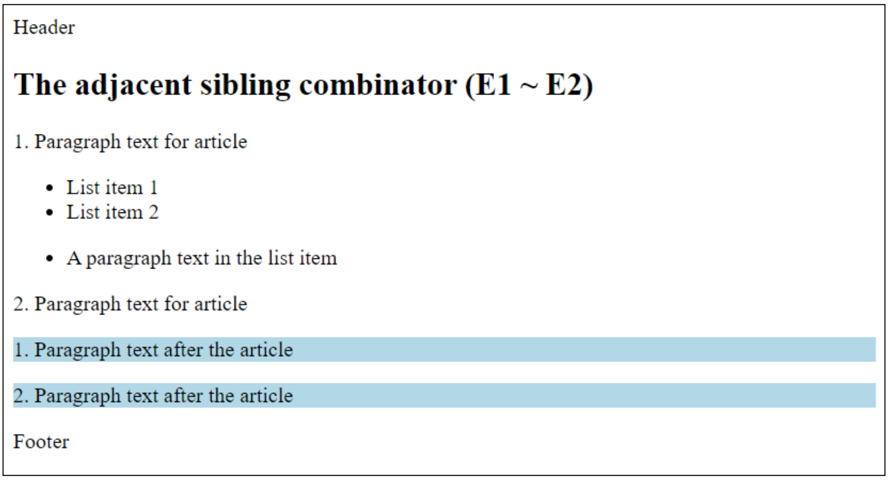


Figure 9.26 General Sibling Combinator Selects All Adjacent <p> Elements



Header

The adjacent sibling combinator (E1 ~ E2)

1. Paragraph text for article

- List item 1
- List item 2

- A paragraph text in the list item

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 9.27 The General Sibling Combinator in Use

Combinator	Name	Meaning
E ~ F	(Indirect) sibling selector (*general sibling combinator*)	F gets selected only if it occurs after E (in the same parent element).

Table 9.16 Overview of the General Sibling Combinator

9.3 Recommendation: How to Use Efficient and Simple CSS

This chapter has introduced a lot of different selectors. At this point, beginners are probably asking themselves what they should use and where. Basically, of course, you can use whatever you want, as long as you achieve your goal. However, using CSS can also make your life easier. In this section, I want to give you a little guide on how to write the most efficient and simple CSS for your projects. Note that this is just a little help to point you in the right direction, not a rule set in stone.

9.3.1 How to Write Well Performing CSS

Modern web browsers can render an HTML document increasingly faster, but that still doesn't mean you shouldn't bother about the website code. This is also true with regard to selecting elements in CSS by means of selectors. Not every selector can access its element in the document structure equally fast.

Do without Combinators If Possible

Without getting too specific here, it can be stated that the simple selectors are usually more efficient than a mixture of multiple selectors with combinators. For example, take a look at the following CSS rule:

```
.myarticle a { color: gray; }
```

Here, `.myarticle a{...}` was used as the descendant combinator. The fact that this selector is slower is due to the way the web browser processes it. If the web browser were to read the document tree from left to right, it would first look for all elements that use the `myarticle` class and then all `a` elements that are a child of it.

But now the web browser reads from *right to left* and first looks for all `a` elements; it's not until after that does it look if there's a matching parent element with the `myarticle` class. For a website with multiple hyperlinks, the web browser would first search for all `a` elements and then filter out again the matching `a` elements that are contained in elements with the `myarticle` class. In that case, it's absolutely unnecessary to select all `a` elements of a website. So, what's wrong with giving the `a` elements in the HTML documents directly a class (e.g., `<a class="myarticle_a">`), which you can then access using a class selector such as `.myarticle_a { color: gray; }`?

I don't mean to disparage the combinators here, as using them for concatenation purposes is extremely powerful, and you can use them to quickly and conveniently style the desired element. However, the descendant combinator is often used unnecessarily in practice. You should use these combinators with a little more care because the web browser has to resolve long and complex combinators first, and this costs the web browser response time when rendering the website.

Admittedly, the selector performance of modern web browsers is pretty good, and some might argue that optimization doesn't play too much of a role here anymore. Nevertheless, sometimes with some thought, you can use a shorter and simpler selector to get where you want to go. Here's another negative example of multiple concatenation of selectors using the descendant combinator:

```
.linklist ul li a { color: green; }
```

Here you're looking for all `a` elements located in a `li` element, which in turn is located in a `ul` element whose parent element contains the `.linklist` class, just to put green font color on it. That's quite some work to be done by the web browser! You could have also written the following:

```
.linklist a { color: green; }
```

This also depends on the HTML document. If there's another ordered list in the HTML document containing an `ol` element that also contains `li` elements with `a` elements in it, you'd have to be more specific, or—even simpler—you could formulate two corresponding classes for it:

```
.linklist-ul { color: green; }
.linklist-ol { color: red; }
```

These optimizations depend on the web project and can't be generalized here. It's just important to think a bit before you implement CSS rules with deeply nested selectors using combinators because there could be an easier way.

Selectors That Don't Make Sense

Selectors are also often used that don't make any sense, which can be avoided with consideration. Here's an example:

```
.myarticle #special a { color: gray; }
```

In this case, you can safely do without the class `myarticle` because an ID selector can only occur once in an HTML document anyway. Consequently, the example can be simplified as follows:

```
#special a { color: gray; }
```

However, again, I don't like the fact that all `a` elements on the website are searched for first, only to be filtered out again in the element with the ID `special`. Again, I'd recommend simplifying this and just using a class selector instead:

```
.special_a { color: gray; }
```

Attention with the Universal Selector

It's probably unnecessary to mention that you should use the universal selector * only in an emergency. After all, this selector selects all elements of an HTML document. Such a negative example could look like the following:

```
#special * { color: black; }
```

Because of the universal selector, all elements of the entire HTML document are searched, and only then are all elements located in `#special`.

But Does It Matter That Much?

Now you've learned how to write a better performing CSS code. Considering the performance of modern web browsers and computers today, some may say that it doesn't matter if the CSS code is fast or not. Unless you're using a website with thousands of elements, the web browser will barely bat an eye when loading a website with inefficient CSS code, and you probably won't notice those few extra milliseconds it takes for the website to load. At this point, one could discuss whether CSS contributes much or little to the loading time of the website.

9.3.2 Recommendation: Keep the CSS Code as Simple as Possible

You still have a guaranteed advantage if you write more efficient CSS: the code will be much neater and simpler in the process, and at the latest when you need to make subsequent changes, you'll be glad you wrote proper CSS. In the previous section, you may have noticed that I've advised using CSS classes for almost every possibility to write better performing CSS. Not only do more precise specifications with multiple selectors act as brakes for the web browser, but they also often lead to more complex CSS, which you then feel in the negative sense when you want to override a specification. In addition, using a type selector such as `h1`, `p`, or `a` alone is often too imprecise and can entail problems when you want to style more-specific elements with additional classes because there are different weightings here. I'll discuss this topic in the following chapter.

For this reason, here's a useful recommendation: work with CSS classes! First of all, this gives you a neat and easy-to-edit CSS, and, secondly, you usually don't have to bother about performance issues—killing two birds with one stone. Of course, this doesn't mean that you have to work exclusively with CSS classes, as there are definitely cases where it makes sense to resort to special selectors or combinations. However, if you pay a little more specific attention to this before creating your website, you should never have much trouble understanding your CSS code after a long time has passed.

9.4 Summary

First, you've learned about the simple selectors of CSS summarized in Table 9.17.

Selector	Name	Selection	HTML Example
`element {...}`	Type selector	HTML element named `element`	`<element>`
`.clname`	Class selector	Element with the `clname` class	`<p class="clname">`
`#elemid`	ID selector	Element with ID `elemid`	`<p id="elemid">`
`*`	Universal selector	All elements	`<p>`
`[attr]`	Attribute selector (presence)	Any element that contains the `attr` attribute	`<element attr="...">`
`[attr^=value]`	Attribute selector (partial value)	Any element where the attribute `attr` starts with the text string `value`	`<elem attr="valueless">`
`[attr$=value]`	Attribute selector (partial value)	Any element where the attribute `attr` ends with the text string `value`	`<elem attr="outvalue">`
`[attr*=value]`	Attribute selector (partial value)	Any element where the attribute `attr` contains the text string `value`	`<elem attr="overvalued">`
`[attr=value]`	Attribute selector (attribute value)	Any element where the attribute `attr` contains the value `value`	`<elem attr="value">`
`[attr~=value]`	Attribute selector (attribute value)	Any element with the value `value` in the attribute `attr` as a standalone word	`<elem attr="abc value xyz">`

Table 9.17 Simple Selectors

Selector	Name	Selection	HTML Example
`[attr|=valx]`	Attribute selector (attribute value)	Any element that has the value `valx` in the `attr` attribute at the beginning as a string separated by a hyphen	`<elem attr="valx-valy">`

Table 9.17 Simple Selectors (Cont.)

For elements that can no longer be reached with simple selectors, you've become acquainted with pseudo-classes. These include the following CSS pseudo-classes `:link`, `:visited`, `:hover`, `:active`, `:focus`, `:placeholder-shown`, `:target`, `:root`, `:empty`, `:first-child`, `:last-child`, `:nth-child()`, `:nth-last-child()`, `:only-child`, `:first-of-type`, `:last-of-type`, `:nth-of-type()`, `:nth-last-of-type`, `:only-of-type`, `:lang()`, and `:not()`.

The CSS pseudo-elements represent another group of selectors. This allows you to address elements that aren't directly present as elements in the structure of the HTML document, but are still resulting from the structure with HTML or are generated using the CSS feature `content`. You should be somewhat familiar with the following pseudo-elements: `::first-letter`, `::first-line`, `::before`, `::after`, and `::selection`.

Finally, you've learned how to concatenate two or more selectors together using combinators. Table 9.18 lists the four combinators available in CSS and described in this chapter.

Combinator	Name	Meaning
`E F`	Descendant selector (*descendant combinator*)	`F` gets selected if it's a descendant of an `E` element.
`E > F`	Child selector (*child combinator*)	`F` gets selected only if it's a direct descendant of an `E` element.
`E + F`	Adjacent sibling selector (*adjacent sibling combinator*)	`F` gets selected only if it occurs directly after `E` (in the same parent element).
`E ~ F`	General sibling selector (*general sibling combinator*)	`F` gets selected only if it occurs after `E` (in the same parent element).

Table 9.18 Various Combinators for Concatenating Selectors

Chapter 10

Inheritance and Cascading

In this chapter, I'll address two other important topics about CSS: the inheritance principle, which refers to the details around defining and passing on CSS features, and the cascading principle, which refers to how a document can be formatted from a variety of stylesheets and different sources.

In this chapter, you'll get to know the details about *inheritance* in CSS, which refers to how, when, and if a CSS feature gets defined and passed on in a central location. In the process, you'll also learn how to force inheritance.

Besides inheritance, *cascading* is an indispensable topic in CSS; after all, the C in CSS stands for *Cascading*. Once you understand the principle of cascading, you can build one stylesheet on top of another and save yourself a lot of typing. The chapter concludes with a digression on how and which values you can pass to CSS features.

10.1 The Principle of Inheritance in CSS

An important principle in CSS is inheritance, which makes it possible to define various CSS features such as color, font, and font size once in a central location instead of assigning the same properties to each individual element again and again.

As you already know, an HTML document is built in a tree structure. The various HTML elements have ancestors and descendants, that is, the parent and child elements. Thanks to this relationship, the subsequent child elements inherit many style properties from the superordinate parent elements.

For a more understandable description of inheritance, you should take a closer look at the following simple HTML document:

```
...
    <link rel="stylesheet" href="css/style.css">
...
  <body>
      <header>Header</header>
      <article>
        <h1>Inheritance</h1>
        <p>1. Paragraph text for article</p>
        <ul>
          <li>List item 1</li>
          <li>List item 2</li>
          <li>List item 3</li>
        </ul>
        <p>2. Paragraph text for article</p>
      </article>
      <p>1. Paragraph text after the article</p>
      <p>2. Paragraph text after the article</p>
      <footer>Footer</footer>
  </body>
...
```

Listing 10.1 /examples/chapter010/10_1/index.html

The `body` element contains a `header` element, an `article` element, two `p` elements, and a `footer` element as direct descendants. This makes the `body` element the parent element of all these elements. Direct descendants of the `article` element, in turn, are the `h1`, `p`, and `ul` elements. These direct descendants of `article` are the indirect descendants (or children's children) of the `body` element.

Related to this example, the following stylesheet should be applied to the document:

```
body {
    background: gray;
    font-family: Arial, Verdana;
    color: white;
}
...
```

Listing 10.2 /examples/chapter010/10_1/css/style.css

Starting from the `body` element, due to inheritance, the CSS features set here are inherited from element to element. All elements contained between `<body>` and `</body>` get the font family Arial (`font-family`) and a white font color (`color`), which was agreed on with the type selector `body {...}`.

Background (Color) Doesn't Get Inherited!

Even if it seems to be the case in Figure 10.2, the gray background color (`background: gray`) doesn't get inherited. The fact that everything here is nevertheless assigned a gray background color is due to the fact that the default value is transparent (translucent). For this reason, the gray color of the `body` element is displayed behind all elements. As a counter-demonstration, you're welcome to use the following universal selector before the `body` selector to recolor all elements to black, so that none of the HTML elements are transparent from scratch anymore, but black:

```
* { background: black; }
body { ... }
```

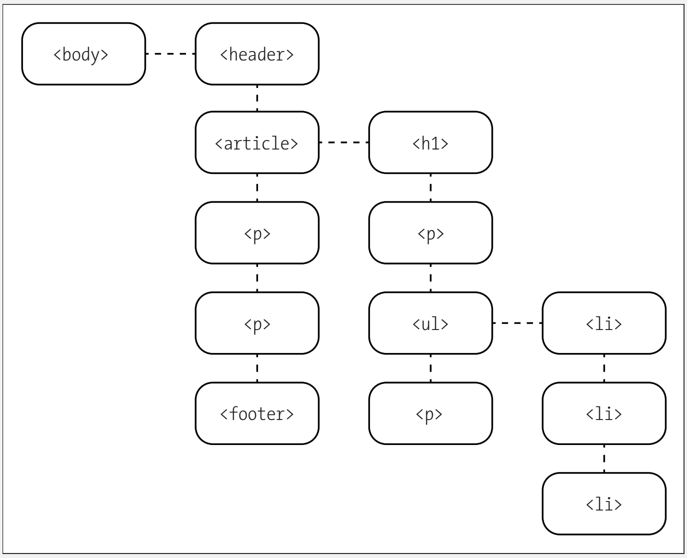
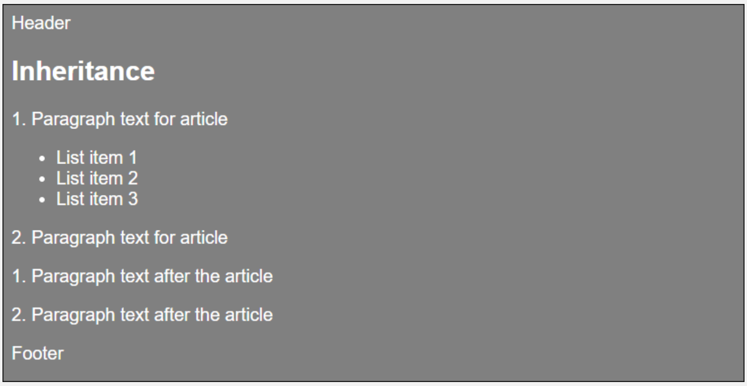


Figure 10.1 Thanks to Inheritance, CSS Features Are Passed on to the Descendants



Header

Inheritance

1. Paragraph text for article

- List item 1
- List item 2
- List item 3

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 10.2 Due to Inheritance, the Text in This Example Is Displayed in White Arial Font with Gray <body> Background

The CSS rule with type selector `body {...}` is extended with the following CSS rule for demonstration purposes:

```
body {
  background: blue;
  font-family: Arial, Verdana;
  color: white;
}
```

```
article {
  background: lightblue;
  color: black;
}
```

Listing 10.3 /examples/chapter010/10_1/css/style.css

If, as in this example with the `article` selector, a new inheritable CSS feature is assigned to an element, such as the text `color` here, the element specified with the selector (here, `article`) and its descendants no longer inherit the CSS features of the parent element. In that case, the CSS features declared in the selector are inherited by the descendants. For this reason, the descendants of an `article` element now get a black text color. The `font-family` of the `body` selector, on the other hand, wasn't declared in the `article` type selector, which is why the font family declared in the `body` selector (here, `Arial`) is still inherited by `article` and its descendants. Again, `background` is *not* passed on to the descendants by the `article` selector, but because of the transparent default background of HTML elements, they are also seen with a light blue background due to the light blue background of the `article` element.

Figure 10.3 shows that the inheritance is valid from the parent element and its descendants onward. The CSS features declared with the `body` selector apply here from `body` to the next element, for which new style properties may have been written that make this element the parent of its descendants (as in the example with the `article` selector).

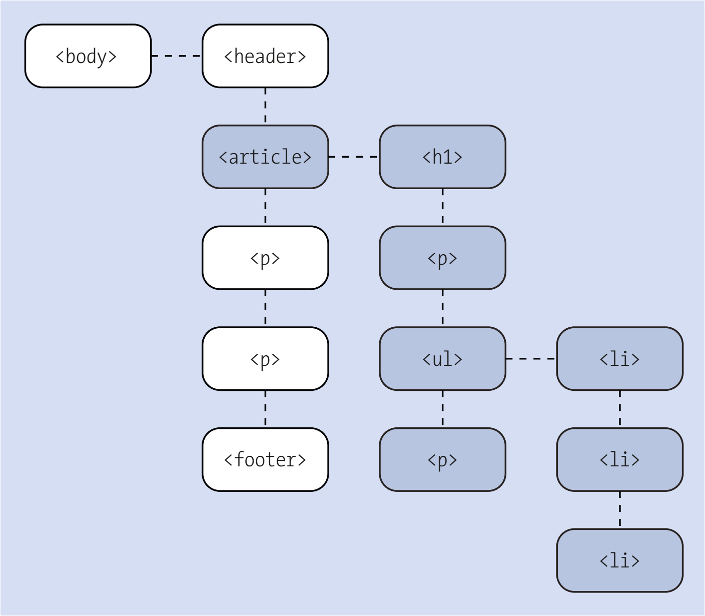
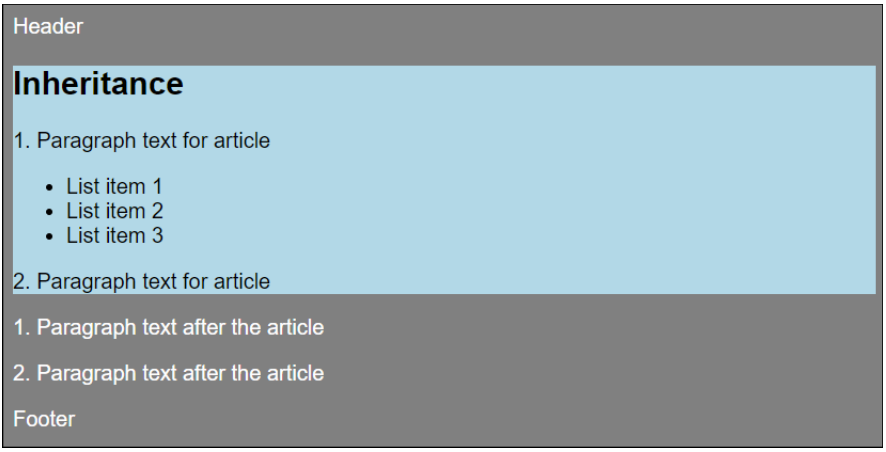


Figure 10.3 Inheritance Applies from the Parent Element to Its Descendants

Default Values for CSS Features

If no specific value has been assigned to a CSS feature, the web browser uses the default value specified for it in the CSS specification when inheriting it.



Header

Inheritance

1. Paragraph text for article

- List item 1
- List item 2
- List item 3

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 10.4 With the “article” Selector, This Element Takes over the Parent Role for the Included Descendants, Styling the Text Color Black

If this inheritance didn’t exist in CSS, you’d have to explicitly create a CSS rule for each element. Inheritance can help you write a more efficient and concise stylesheet. Often, for example, it’s sufficient to set the font and other CSS features fairly early in the `body` element, and the inheritance rule makes sure that you don’t have to repeat those CSS features later for each element.

Use Inheritance as Much as Possible

If you master inheritance and use it consciously, you can do without the odd selector or the multiple setting or changing of CSS features. For example, if you’ve set the font with the CSS feature `font-family` or the font size with `font-size` for `body`, this applies to the entire document, and you don’t need to do that again separately for `p` or `li` elements. Of course, this also has a positive effect on performance.

10.1.1 Be Cautious When Using Relative Properties

The inheritance of relative units such as `font-size` with percentage values (`font-size: 90%`) or `em` (`font-size: 0.9em`), surprising changes may occur because some web browsers reapply this value for each element if a `font-size` with percentages or `em` is also defined for further elements. For example, let's say you set CSS rules like the following:

```
body { font-size: 70%; }
...
p { font-size: 70%; }
```

This would result in the content of the `p` element displaying only a font size of 49% due to the inheritance of the relative value from the `body` element because such relative values would be applied cumulatively to each element. It's not the value defined in the stylesheet that's passed on, but the value calculated by the web browser.

10.1.2 Not Everything Gets Inherited

Not all CSS features are inherited by the descendants, as you've already experienced previously with the `background` property. Especially with CSS features such as `margin`, `width`, `padding`, or `border` it wouldn't make much sense and more likely give you a headache than help you. For example, if the CSS feature `border` (for a border around an element) were passed on to each descendant, you would have to deactivate this inheritance using `border: none` for each descendant. In anticipation of this, here's a brief list of some characteristics that don't get inherited. These common values aren't inherited:

- `background`, `border` (see Chapter 11, Section 11.5)
- `width`, `height`, `padding`, `border`, `margin` (see Chapter 11, Section 11.1)
- `position`, `top`, `right`, `bottom`, `left` (see Chapter 12, Section 12.1)
- `float` and `clear` (see Chapter 12, Section 12.3)

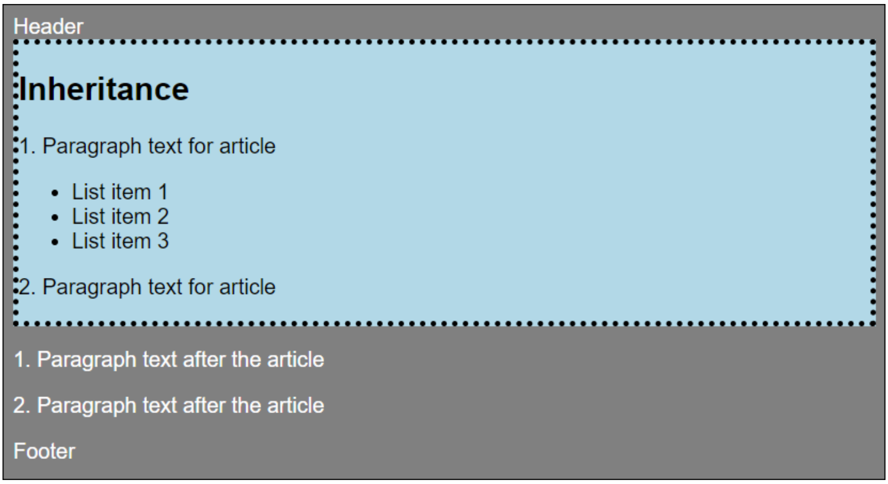
10.1.3 Enforcing Inheritance Using "inherit"

As mentioned in the previous section, there are CSS features that don't get inherited. If the situation requires it, you can force inheritance using the `inherit` keyword. To better demonstrate the `inherit` keyword, let's first look at an example without the keyword:

```
body {
  font-family: Arial, Verdana;
  color: white;
  background: gray;
}

article {
  border: 4px dotted black;
  background: lightblue;
  color: black;
}
...
```

Listing 10.4 /examples/chapter010/10_1_3/css/style.css

Based on this example, these few lines of CSS provide the following picture:



Header

Inheritance

1. Paragraph text for article

- List item 1
- List item 2
- List item 3

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 10.5 Not the Result We Wanted

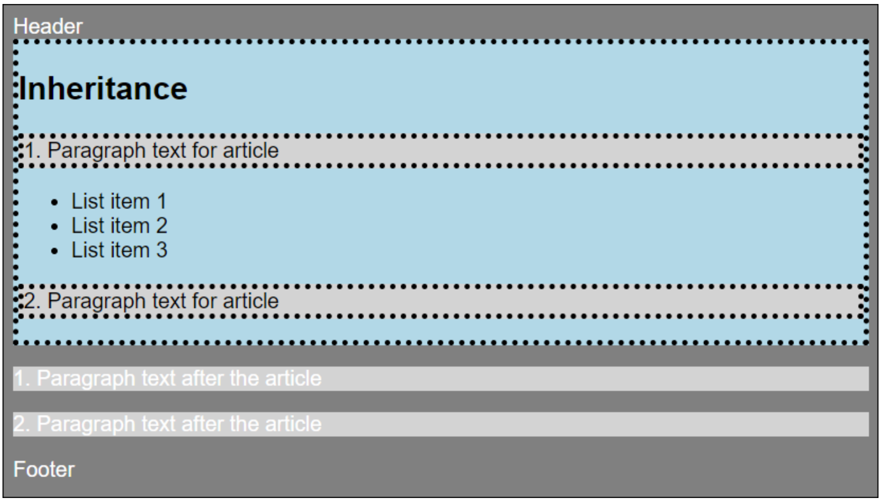
If, in this example, you want to force all `p` elements inside `<article>` to also have the dotted black border, you can force this with `inherit` by adding the following CSS rule:

```
...
article {
  border: 4px dotted black;
  background: lightblue;
  color: black;
}

p {
  border: inherit;
  background: lightgray;
}
```

Listing 10.5 /examples/chapter010/10_1_3/css/style.css

By assigning `inherit` to the CSS feature, you force all `p` descendants following in the `article` element to inherit the `border` of the parent element (here, `article`). The result is shown in Figure 10.6.

Here you can also see that the inheritance only affects the child elements, so `inherit` really only changed the `border` values of `p` elements whose parent element was an `article` element. Consequently, no border was added to the other two `p` elements outside the `article` element, but the background color was changed. Thus, it's important to understand that with `inherit`, a child element can inherit only the values of the parent element.



Header

Inheritance

1. Paragraph text for article

- List item 1
- List item 2
- List item 3

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 10.6 Inheritance Forced via "inherit"

10.1.4 Restoring the Default Value of a CSS Feature ("initial")

When you assign the `initial` value to a CSS feature, the default value (or starting point) of the property specified in the CSS specification will be used. The value `initial` can be assigned to any CSS feature. Referring to Figure 10.6, if you want to reset the font color of the `p` elements to the default value of the web browser (which is usually black), you can do it as follows:

```
...
p {
  border: inherit;
  background: unset;
  color: initial;
}
...
```

Listing 10.6 /examples/chapter010/10_1_4/css/style.css

Here, you can use `color:initial;` to make sure that the font color of all `p` elements in the HTML document is reset to the default value of the web browser.

10.1.5 Forcing Inheritance or Restoring a Value ("unset")

The `unset` keyword gives you a middle ground between `inherit` and `initial`. When you use the keyword for a CSS feature, it behaves like the `inherit` keyword and inherits the value for the corresponding CSS feature of the parent element. If there's no parent element here with a set value for this CSS feature, this keyword behaves like `initial` and resets a CSS feature to the default value given by the CSS specification.

10.1.6 Forcing Inheritance or Restoring Values for All Properties

Finally, there's the property `all`, which is a short notation for inheriting all CSS features of the parent element with `inherit` or resetting them to the default value with `initial`. The `unset` value described previously can also be used along with `all`. Consider the following example:

```
...
      <p>1. Paragraph text after the article</p>
      <p class="p_outside">2. Paragraph text after the article</p>
      <footer>Footer</footer>
...
```

Listing 10.7 /examples/chapter010/10_1_6/index.html

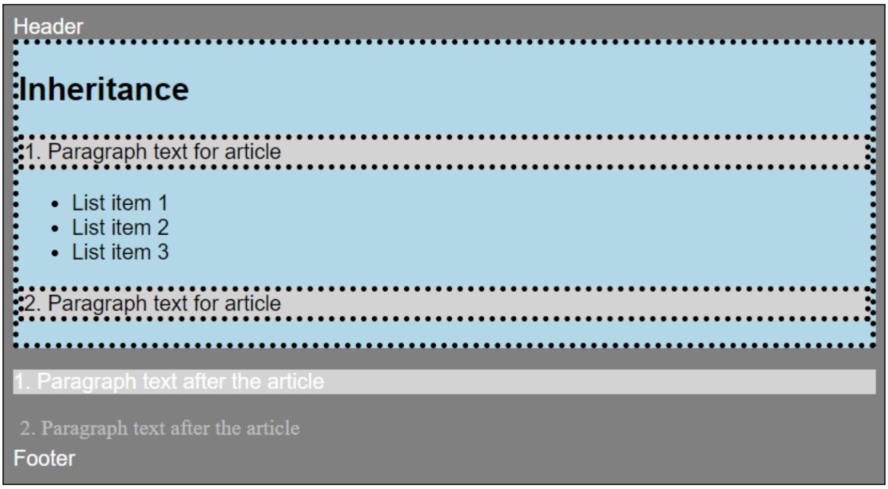
With regard to our example, here I used a `p_outside` class for the second `p` element. For demonstration purposes, I want to reset and restyle this element with `initial`:

```
body {
  font-family: Arial, Verdana;
  color: white;
  background: gray;
}

article {
  border: 4px dotted black;
  background: lightblue;
  color: black;
}

p {
  border: inherit;
  background: lightgray;
  color: unset;
}

.p_outside {
  all: initial;
  display: block;
  margin: 5px;
  color: silver;
}
```

Listing 10.8 /examples/chapter010/10_1_6/css/style.css

Here, the paragraph text of the `p` element after the `article` element was covered with a white font and light gray background. Because the paragraph text with the `p` element above it should still remain in this format, I wrapped the new style in a `p_outside` class, reset all CSS features using `all: initial;`, and then began to restyle the paragraph text.

Note, however, that the CSS feature `all` was used only for demonstration purposes and isn't intended for styling entire elements without any further consideration, only to reset everything again. If you happen to find yourself in the situation that you have to use this feature more often than necessary, you may want to rethink the way you use CSS.



Header

Inheritance

1. Paragraph text for article

- List item 1
- List item 2
- List item 3

2. Paragraph text for article

1. Paragraph text after the article

2. Paragraph text after the article

Footer

Figure 10.7 Using a Class, I Set All the CSS Features for the Second <p> Element outside the <article> Element to the Default Value with “all: initial;” and Restyled It

10.2 Understanding the Control System for Cascading

The term cascading means that a document can be formatted not only by one stylesheet, but from a variety of stylesheets that can come from different sources. This makes it possible for one stylesheet, if used correctly, to build on another, saving you a lot of work. Now, the fact is that the many ways to include and combine stylesheet statements can cause conflicts and contradictions. Such a conflict arises when the same CSS feature has been assigned different values in multiple statements. For such cases, there's a system of rules that decide which of the conflicting or competing style statements will ultimately be applied to an element.

10.2.1 The Origin of a Stylesheet

A stylesheet can originate from three different sources—web browser, user, and author—as described here:

- **Browser stylesheet**
  The default stylesheet of the web browser is used if no CSS formatting is assigned to the HTML document. Every web browser provides basic formatting for this purpose, with default values for font sizes for headings, underlining for links, indentation for lists, spacing for paragraphs, and much more. Each web browser provides its own default setting, so there are usually slight differences.
- **User stylesheet**
  Some web browsers allow users to include their own stylesheet files directly or via extensions. If they include their own stylesheets (user stylesheets, user styles), the corresponding properties of the web browser will be overwritten.
- **Author stylesheet**
  This is the stylesheet you created and referenced or included with the `@import` rule, which is usually used to override or combine various CSS features of the default stylesheet—the browser stylesheet)—and the user stylesheet, if applicable.

Among these three sources, the browser stylesheet has the lowest priority. If a user stylesheet is used—which has a medium priority—the browser stylesheet will be overwritten. The author stylesheet, on the other hand, has the highest priority and overrides both the browser and user stylesheets. Simply put, these three sources offer you three ways to apply CSS features to an HTML element:

- If no CSS feature is declared for an HTML element, then a check is carried out if anything is inherited. If nothing is inherited, the default value of the browser stylesheet will be used.
- If the web browser finds exactly *one* declaration of a CSS feature for a particular element, it uses the declaration to format the HTML element.
- On the other hand, if the web browser finds *multiple* identical and competing declarations of a CSS feature for a given HTML element, it sorts them according to their importance, specificity, and order, and uses them accordingly.

Basically, this principle of three sources is kept relatively simple here, and if the web browser finds no or only one CSS feature for an HTML element, things are relatively clear. It gets more difficult when several competing declarations coincide. You'll learn how CSS solves this problem on the following pages.

10.2.2 Increasing the Priority of a CSS Feature Using "!important"

If you declare a CSS rule or CSS feature multiple times in the same file, usually the last property declared gets the nod. Let's take a look at a simple example:

```
.m_article { border: 1px solid gray; }
...
.m_article { border: 1px dotted green; }
```

Here, an `m_article` class was initially formatted with a solid gray border that has a thickness of 1 pixel. A bit further down there's another CSS rule for this, but now `m_article` is formatted with a dotted green border whose thickness is 1 pixel. The last declaration noted gets the deal.

With the CSS keyword `!important`, on the other hand, you can increase the priority so that this property can't be overridden by the subsequent specifications. You can apply this keyword to the example just shown as follows:

```
.m_article { border: 1px solid gray !important; }
...
.m_article { border: 1px dotted green; }
```

In this case, the declaration made last is no longer given preference. In this example, the `!important` keyword draws a solid gray border with a thickness of 1 pixel around the element containing the `m_article` class.

10.2.3 Sorting by Importance and Origin

The `!important` keyword slightly changes the priority order, which used to be *author stylesheet before user stylesheet before browser stylesheet*. Sorting by importance can be written in the following order of descending priority (importance):

1. User stylesheet with `!important`
2. Author stylesheet with `!important`
3. Author stylesheet
4. User stylesheet
5. Browser stylesheet

In practice, you'll probably be dealing with your own author stylesheet most of the time (without `!important`).

You might wonder why the user stylesheet with `!important` is suddenly more important than the author stylesheet with `!important` because without `!important`, it's the other way around. Thus, if a user marks a CSS feature with `!important` in the stylesheet, it has to be accepted by the web browser, no matter what you've set there.

The reason that a user stylesheet with `!important` has a higher importance than an author stylesheet with `!important` is that people with a disability should be preferred, so that they may be able to determine by themselves in what way the content should be displayed. For example, a person with a visual impairment could increase the paragraph font size using a custom stylesheet as follows:

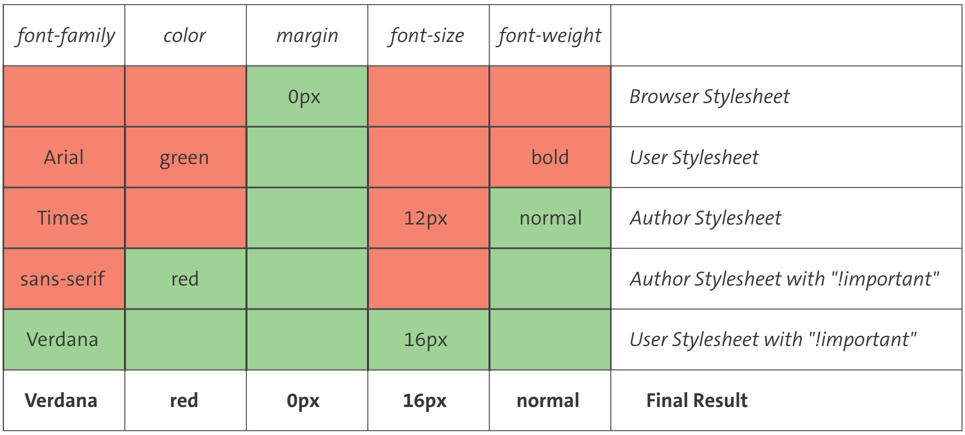
```
p { font-size: 200% !important; }
```

In practice, you shouldn't expect every visitor with an impairment to be familiar with CSS, let alone be able to create their own CSS stylesheet. In that case, you could offer a suitable stylesheet for download.

Integrating User Stylesheets

The integration of user stylesheets differs slightly from web browser to web browser. In some web browsers, you'll find this option hidden in the submenus or settings, while other web browsers ask for an extension to be installed first. An interesting project related to this topic can be found at *http://userstyles.org*, where you can install ready-made user stylesheets for Facebook, Google, Tumblr, and other high-traffic websites using the *Stylish* extension for the Firefox and Chrome web browsers.

Figure 10.8 demonstrates such a sorting according to the source and its importance or simply the cascading flow for a `p` element, for example.



font-family	*color*	*margin*	*font-size*	*font-weight*	
		0px			*Browser Stylesheet*
Arial	green			bold	*User Stylesheet*
Times			12px	normal	*Author Stylesheet*
sans-serif	red				*Author Stylesheet with "!important"*
Verdana			16px		*User Stylesheet with "!important"*
Verdana	**red**	**0px**	**16px**	**normal**	**Final Result**

Figure 10.8 Theoretical Example of Sorting by Importance

CSS Features for the Appropriate Media Type

I haven't mentioned yet that in the first step, before sorting by importance, the system checks whether CSS features have been included for an HTML element that are also valid for the current media type. The first step is to search for all declarations that should be applied to the current output medium for the elements.

10.2.4 Sorting by Weighting the Selectors (Specificity)

Besides sorting the importance according to the origin of the stylesheets, there's also a priority rule among the selectors. This type of sorting is used when there are equivalent specifications within a stylesheet. In this process, a value is calculated for each selector, indicating the weighting of the selector. That value is referred to as the *specificity*. The specificity is expressed as a numerical value, and the higher this numerical value, the more important the selector, which then overrides any competing selector with a lower value.

The specificities of a selector are divided into the following four groups:

- **A** (first digit: **1**,0,0,0)
  This value is set only if the HTML element uses inline styling via the `style` attribute. Otherwise, this value remains at 0.
- **B** (second digit: 0,**1**,0,0)
  This group counts the number of ID attributes in a selector.
- **C** (third digit: 0,0,**1**,0)
  This group stores the number of attributes, classes, and pseudo-classes selected by a selector.
- **D** (fourth digit: 0,0,0,**1**)
  The group with the lowest weighting contains the number of element names (type selectors) and pseudo-elements.

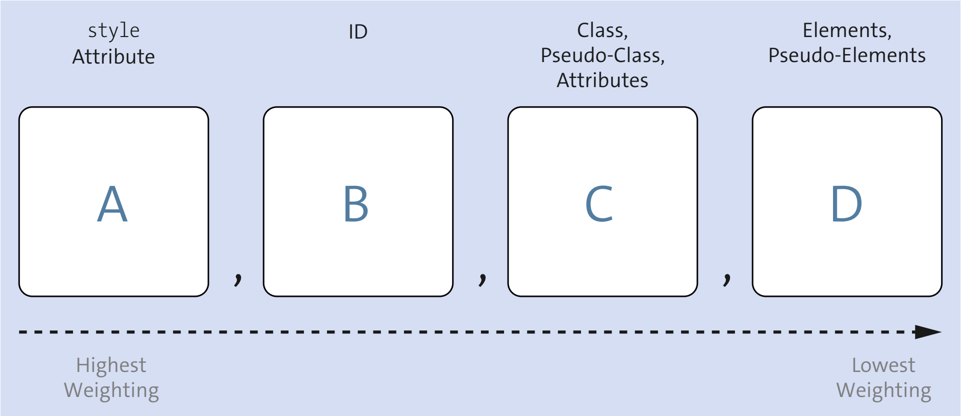


Figure 10.9 Calculating the Specificity: If There's a Conflict, the Web Browser Will Use the Selector with the Higher Weighting

No Weighting for...

Universal selectors with * don't get any weighting and behave neutrally. The same applies to the *combinator characters* >, +, and ~, as well as the space between two selectors. The two selectors do increase the weighting in group D by at least the value 2 (because there are two selectors), but the combinator characters are also neutral here.

HTML elements or attributes classified as *deprecated* by the W3C are also evaluated by current web browsers without a weighting.

In addition, the pseudo-class `:not()` doesn't add any weighting. For this purpose, the elements within the pseudo-class are evaluated as prescribed.

Table 10.1 contains several examples of such calculations.

Selectors	A	B	C	D	Specificity	Description
`*{...}`	0	0	0	0	0,0,0,0	1 universal selector
`p{...}`	0	0	0	1	0,0,0,1	1 type selector
`ol li{...}`	0	0	0	2	0,0,0,2	2 type selectors
`.note`	0	0	1	0	0,0,1,0	1 class selector
`*.note`	0	0	1	0	0,0,1,0	1 universal selector 1 class selector
`*[type=checkbox]{...}`	0	0	1	0	0,0,1,0	1 universal selector 1 attribute selector
`p:first-child{...}`	0	0	1	1	0,0,1,1	1 pseudo class 1 type selector
`ul li.info{...}`	0	0	1	2	0,0,1,2	2 type selectors 1 class selector
`#content{...}`	0	1	0	0	0,1,0,0	1 ID selector
`#content p{...}`	0	1	0	1	0,1,0,1	1 ID selector 1 type selector
`#content *:not(nav) li{...}`	0	1	0	2	0,1,0,2	1 ID selector 1 universal selector 2 type selectors 1 pseudo-class (`:not()` doesn't count)
`ul#nav li.hyper a{...}`	0	1	1	3	0,1,1,3	3 type selectors 1 class selector 1 ID selector
`<a style="...">`	1	0	0	0	1,0,0,0	1 inline style

Table 10.1 Some Sample Calculations of the Specificity of Selectors

When you look at the end result of specificity in Table 10.1, you could also imagine a specificity such as 0,0,1,1 as decimal 11 or specificity 0,1,1,3 as decimal 113.

Nevertheless, you should keep in mind that this is *not* the decimal system with base 10. Consequently, a commonly heard explanation such as “ID selector counts 100, class selector counts 10, and normal element counts 1” isn’t correct. For example, if you have a specificity such as 0,1,2,11 (exaggerated), this does *not* mean that 11 for category D results in the decimal value 131, as it would in the decimal system with base 10. Admittedly, the value 11 for group D is a bit exaggerated; this would be the case, for example, if you used 11 type selectors there.

Let’s look at a simple example:

```
...
<article>
  <h1>Inheritance</h1>
  <p>1. Paragraph text for article</p>
  <ul id="index">
    <li class="aclass">list item 1</li>
    <li class="aclass">list item 2</li>
  </ul>
</article>
...
```

Listing 10.9 /examples/chapter010/10_2_4/index.html

From this HTML code, the `li` element is to be selected using the following CSS rules, all of which compete with each other:

```
.aclass { color: green; }
#index li.aclass { color: orange; }
li { color: red; }
li.aclass { color: blue; }
body article ul li { color: yellow; }
#index li { color: gray; }
```

Listing 10.10 /examples/chapter010/10_2_4/css/style.css

You’re now invited to try out for yourself in which color the `li` element will eventually be displayed, or use the specificity of selectors learned previously and calculate for yourself which CSS rule has the highest weighting. The solution is shown in Table 10.2.

Selectors	A	B	C	D	Specificity	Description
`.aclass`	0	0	1	0	0,0,1,0	1 class selector

Table 10.2 Calculating the Specificity of Selectors; in the Example, the <li> Element Is Shown in Orange

Selectors	A	B	C	D	Specificity	Description
`#index li.aclass`	0	1	1	1	**0,1,1,1**	1 ID selector 1 type selector 1 class selector
`li`	0	0	0	1	0,0,0,1	1 type selector
`li.aclass`	0	0	1	1	0,0,1,1	1 type selector 1 class selector
`body article ul li`	0	0	0	4	0,0,0,4	4 type selectors
`#index li`	0	1	0	1	0,1,0,1	1 ID selector 1 type selector

Table 10.2 Calculating the Specificity of Selectors; in the Example, the <li> Element Is Shown in Orange (Cont.)

What about the Short Notation "h1, h2, h3, p {…}"?

If you separate multiple type selectors with commas, such as the following:

```
h1, h2, h3, p { font-family: Arial; }
```

Those four selectors will only be counted in total with the value 1 in group D. This notation ultimately corresponds only to a shorter notation:

```
h1 { font-family: Arial; }
h2 { font-family: Arial; }
h3 { font-family: Arial; }
p { font-family: Arial; }
```

Sorting with Equal Weighting

If two CSS rules have the same weight and are of the same origin, then the rule that occurred last takes precedence. However, note that CSS features declared with `!important` again take precedence. The only way to override a CSS feature declared with `!important` is to use another `!important` declaration. In terms of specificity, you could imagine `!important` with the value 1,0,0,0—although this category or group doesn't exist in CSS.

10.2.5 Summary of the Cascading Rules System

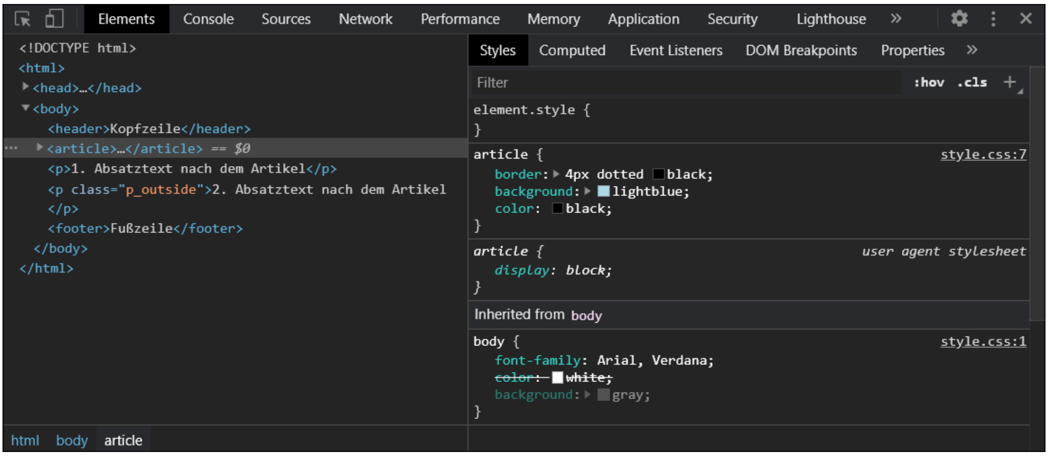
Here's another summarizing overview of how the cascading sequence is processed in a web browser when it searches for stylesheet specifications in a document:

1. Sorting by origin and importance in the following order:
   - User stylesheet with `!important` goes before author stylesheet with `!important`.
   - Author stylesheet with `!important` has priority over author stylesheet without `!important`.
   - Author stylesheet without `!important` takes preference over user stylesheet without `!important`.
   - User stylesheet without `!important` takes preference over browser stylesheets.
2. For equivalent specifications within stylesheets, in turn, the specificity gets calculated and, according to the weighting, the selector with a higher calculated value is used.
3. For specifications that contain an identical specificity, the most recently occurred statement takes precedence.

10.2.6 Analyzing the Cascading in the Browser

Again, the developer tools of the web browser can be of great help. In most web browsers, you open these tools via `Ctrl` + `Shift` + `I`. If you select the styled HTML element here on the left, the CSS for it gets displayed in the **Styles** tab on the right. There you can see which CSS features are inherited from the web browser (**User agent stylesheet**) and which are inherited from other elements (**Inherited from**). Elements that are crossed-out were overwritten by another element. For example, in Figure 10.10 the CSS feature `color` of the `body` element was overridden in the `article` element.

The analysis via the developer tool of the web browser is indispensable for the design of a website and for finding an error if it doesn't work properly with the CSS of a certain element.



```
<!DOCTYPE html>
<html>
  <head>…</head>
  <body>
    <header>Kopfzeile</header>
    <article>…</article> == $0
    <p>1. Absatztext nach dem Artikel</p>
    <p class="p_outside">2. Absatztext nach dem Artikel
    </p>
    <footer>Fußzeile</footer>
  </body>
</html>
```

```
element.style {
}
article {                                   style.css:7
   border: 4px dotted black;
   background: lightblue;
   color: black;
}
article {                       user agent stylesheet
   display: block;
}
Inherited from body
body {                                      style.css:1
   font-family: Arial, Verdana;
   ~~color: white;~~
   background: gray;
}
```

Figure 10.10 Indispensable for Analyzing the CSS Is the Developer Tool of the Web Browser

10.3 Related Topic: Passing Values to CSS Features

To conclude the chapter, you'll learn a few things about passing values to CSS features. I'll limit the individual descriptions to what is most necessary. More details about where you can use which value and how will be described in the remaining CSS chapters when the respective CSS feature is covered.

Further Online Reading

For more information on CSS value specifications and some other allowed value types, visit the W3C website at *www.w3.org/TR/css-values-3/*. At this point, I'll only describe the most common types, which you'll encounter more frequently in the following chapters.

10.3.1 Different Units of Measurement in CSS

The specification of numerical values, for example, for font sizes, heights, and widths or for distances, is often written in connection with a unit of measurement (UoM) directly after the numerical value. In CSS, there are many UoMs that can be used as either relative or absolute specifications. For the number 0, the UoM doesn't have to be specified. Here are some examples of length specifications with different UoMs:

```
font-size: 12pt;
margin-left: 1em;
line-height: 125%;
border-width: 0.2in;
```

Specifying Numerical Values or Numbers

The numeric values in CSS can be integers and floating-point numbers. The comma in the floating-point number is represented by a dot in CSS (e.g., `font-size: 1.2em`). There are also numbers in CSS that are used without a UoM (e.g., `z-index: 1` or `opacity: 0.75`). Negative values with a minus sign can also be used if this seems reasonable (e.g., `z-index: -2`). Negative values can't be used for length specifications such as `height` and `width`, or for `padding`.

See Table 10.3 for a list of common UoMs for CSS. In the further course of the book, you'll learn in more detail where you can or should use which UoM.

UoM	CSS Name	Specification	Description
Pixel	`px`	Absolute relative	The display of pixels depends on the pixel density of the output device. With a high screen resolution, the pixels become smaller, which is why the display appears smaller. It's the same with a low screen resolution, where the pixels and hence the display appear larger. A pixel is thus a relative UoM on display devices and an absolute UoM in relation to the display or content.
Point	`pt`	Absolute	This is a typographic UoM, and 1 point is equal to 1/72 inch, for example: `font-size: 14pt;`
Pica	`pc`	Absolute	Pica is also a UoM widely used in typography. A pica is 1/6 of an inch and is therefore equivalent to 12 points, for example: `font-size: 1pc; /*= 12pt */`
Centimeters	`cm`	Absolute	A centimeter is the hundredth part of a meter and corresponds to 10 mm, for example: `margin-left: 1.3cm;`
Millimeters	`mm`	Absolute	A millimeter is one-thousandth of a meter or the tenth part of a centimeter, for example: `padding: 2mm;`
Inches	`in`	Absolute	1 inch is equal to 2.54 cm, for example: `margin-top: 1in;`
em quad	`em`	Relative	An `em` represents the font size of the element. If `em` is used for the font size itself (`font-size`), then this value refers to the font size of the parent element, for example: `font-size: 1.3em;`
x-height	`ex`	Relative	An `ex` represents the height of the lowercase x of the font used in this element. Again, if `ex` is used for the `font-size`, then this height refers to the value of the lowercase x in the parent element.

Table 10.3 Common Absolute and Relative Length Specifications in CSS

UoM	CSS Name	Specification	Description
Percent	`%`	Relative	Percentages can be used in a variety of ways. It depends on the CSS feature whether this value refers to the element's own size, to that of the parent element, or to a general context.
Root em	`rem`	Relative	The `rem` (`rem` = *root em*) behaves in the same way as `em`, except that the `rem` value is based on the root element and not on the font size of the respective parent element. In HTML, the root element is the `body` or `html` element, for example: `font-size: 1.2rem;`
Viewport width	`vw`	Relative	`1vw` corresponds to 1% or the hundredth part of the width of the display area (i.e.,viewport). Thus, `100vw` corresponds to the complete width of the display area. This UoM allows you to adjust the font size to the display area of the device or the size of the browser window.
Viewport height	`vh`	Relative	`1vh` corresponds to 1% or the hundredth part of the height of the display area (i.e., viewport). Thus, `100vh` corresponds to the total height of the display area. This UoM allows you to adjust the font size to the display area of the device or the size of the browser window.

Table 10.3 Common Absolute and Relative Length Specifications in CSS (Cont.)

10.3.2 Character Strings and Keywords as Values for CSS Features

CSS makes a strict distinction between keywords and strings. Strings are enclosed between single or double quotes in CSS. Here's an example with strings in CSS:

```
content: " meter";  /* string */
content: '$ ';      /* string */
```

The keywords in CSS, on the other hand, aren't marked separately, for example:

```
color: black;
width: auto;
display: inline-block;
```

You can't just put keywords between single and double quotes. A statement the following would have no effect:

```
color: "black"; /* !!! string !!! CSS error */
```

Here the color wouldn't be set to black because you made it an ordinary string by using double quotes. Strictly speaking, this is an invalid value assignment, which a CSS validator would find fault with if you had this line checked.

10.3.3 Many Ways of Using a Color in CSS

Colors generally represent an important design element. In CSS, you have several options for writing color values. It should be noted right away that none of the different color specifications has any advantages or disadvantages in the display or execution of web pages. The only advantages or disadvantages are probably more for the individual who may not be familiar with hexadecimal notation and thus can use named colors. Graphic designers who are familiar with RGB or HSL colors will find these color specifications easier to use. Consequently, there's something for everyone.

Named Colors Using a Name as a Color Value

Even at the time CSS was introduced, it was possible to write color values directly with a name. For this purpose, 16 VGA color names were initially added to the CSS specification (later, orange was added). The advantage of such color names is, of course, that you can easily remember and use them (at least those of the basic colors), for example:

```
.p_article {
  background-color: blue;
  color: white;
}
```

Here, a CSS rule has been defined for a paragraph text with the `<p>` element with blue background and white font. In Table 10.4, you'll find classic color names that have been part of CSS since its beginnings.

CSS Name	Hexadecimal	RGB	Color
black	#000000	rgb(0,0,0)	Black
gray	#808080	rgb(128,128,128)	Gray
silver	#C0C0C0	rgb(192,192,192)	Silver
white	#FFFFFF	rgb(255,255,255)	White
purple	#800080	rgb(128,0,128)	Purple

Table 10.4 Classic Named Colors

CSS Name	Hexadecimal	RGB	Color
`fuchsia`	`#FF00FF`	`rgb(255,0,255)`	Fuchsia
`maroon`	`#800000`	`rgb(128,0,0)`	Maroon
`red`	`#FF0000`	`rgb(255,0,0)`	Red
`olive`	`#808000`	`rgb(128,128,0)`	Olive
`yellow`	`#FFFF00`	`rgb(255,255,0)`	Yellow
`green`	`#008000`	`rgb(0,128,0)`	Green
`lime`	`#00FF00`	`rgb(0,255,0)`	Lime
`navy`	`#000080`	`rgb(0,0,128)`	Navy
`blue`	`#0000FF`	`rgb(0,0,255)`	Blue
`teal`	`#008080`	`rgb(0,128,128)`	Teal
`aqua`	`#00FFFF`	`rgb(0,255,255)`	Aqua
`orange`	`#FFA500`	`rgb(255,165,0)`	Orange

Table 10.4 Classic Named Colors (Cont.)

In the current CSS, additional color values have been added to these named ones, which are now understood by all popular web browsers. There should be 147 color names by now, although it's also permitted to write all color names containing the word `gray` with e (`grey`). You can also find a list of color names, neatly sorted, at *www.colors.commutercreative.com*.

Classic Hexadecimal Notation for the Color Specification

If you look at the HTML code of web pages, the use of hexadecimal notation still seems to be the most commonly used notation for color values. The specification starts with the `#` character, followed by the color components for red, green, and blue in a range from `00` (for 0) to `FF` (for 255). The general notation of the hexadecimal notation is

```
#RRGGBB
```

Here `R` stands for the red portion, `G` for the green, and `B` for the blue one. Let's take a look at a simple example:

```
.p_article {
  background-color: #0000FF;
  color: #FFFFFF;
}
```

Here again, a CSS rule was defined for a paragraph text containing the `<p>` element with a blue background and white font. Table 10.4 contains an overview of different color specifications with the corresponding hexadecimal values. In contrast to named color names, where you're limited to a certain number and fixed defaults of color names, with such RGB mixing, you can theoretically reproduce 256 × 256 × 256 colors (over 16.7 million) colors.

Short Hexadecimal Notation

In CSS, you can abbreviate the hexadecimal notation if the first and second digits of red, green, or blue are identical. The specification `#0F0` thus corresponds to `#00FF00`, or the short notation `#123` corresponds to `#112233`. The web browser converts the `#RGB` value to the six-digit `#RRGGBB` value, for example:

```
.p_article {
  background-color: #00F;
  color: #FFF;
}
```

This is identical to the following:

```
.p_article {
  background-color: #0000FF;
  color: #FFFFFF;
}
```

The FF value is the hexadecimal notation for the value 255. In contrast to the decimal system with base 10, the hexadecimal system uses base 16. Here, the characters A to F are used from the value 10 onward (see Table 10.5).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	100	255
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	64	FF

Table 10.5 Comparison between the Counting of the Decimal System with Base 10 (Upper Row) and the Hexadecimal System with Base 16

Mixing the Colors with Red, Green, and Blue (RGB Mixture)

You can also define an RGB color mixture using the CSS function, `rgb(Red, Green, Blue)`. You make the specifications for red, green, and blue either with percentage values (0–100%) or with decimal numbers in the range of 0–255. An example of using the `rgb()` function might look as follows:

```
.p_article {
  background-color: rgb(0,0,255);
  color: rgb(255,255,255);
}
```

The following is an example as a percentage value:

```
.p_article {
  background-color: rgb(0%,0%,100%);
  color: rgb(100%,100%,100%);
}
```

Again, in both examples, you've used the class selector to define a CSS rule for an HTML element of the `p_article` class with a blue background and white font. In Table 10.4, you'll find various color specifications with the corresponding RGB mixtures for the `rgb()` function.

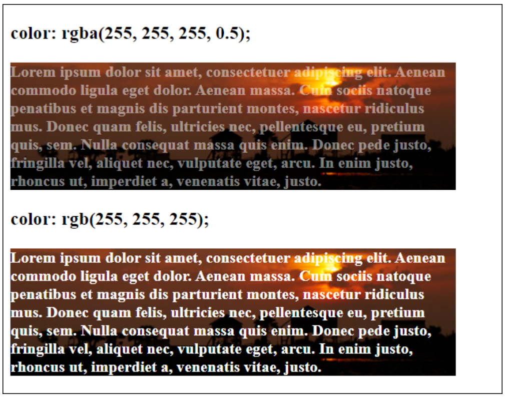
RGB Mixture with Transparency

Another CSS function is `rgba()`, which adds a fourth value—the alpha value for transparency—to the ordinary `rgb()` function. This value allows you to specify the transparency for the color. The value `0.0` represents full transparency, while the value `1.0` stands for full opacity and corresponds to the function, `rgb()`.

With the following specifications, you specify that the paragraph text in white color with `0.3` is exactly 30% visible and 70% transparent:

```
.p_article {
  background-image: url('background.jpg');
  background-repeat: no-repeat;

  color: rgba(255, 255, 255, 0.3);
}
```

Here, an image has also been inserted with `background-image` to demonstrate how the image remains visible through the text thanks to the `rgba()` function. You can fine-tune the transparency value even more and use a second decimal place (hundredths) after the decimal point, for example, `0.35`.



color: rgba(255, 255, 255, 0.5);

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

color: rgb(255, 255, 255);

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

Figure 10.11 Using "rgba()"

An example of this is shown in Figure 10.11. The paragraph text was created over the top background image using the `rgba()` function in white font with a transparency of 50%. In the paragraph text below, the familiar `rgb()` function was used. The example can be found in */examples/chapter010/10_3_3/index.html*.

HSL Mixture: Mixing Colors with Hue, Saturation, and Lightness

Not everyone likes the RGB color mix with red, green, and blue, and not everyone is familiar with it. For this purpose, as of Level 3 CSS, it provides a mixture of Hue, Saturation, and Lightness (HSL), which is referred to as `hsl`.

Many web designers find it more intuitive or easier to remember to specify the hue with an integer value from 0 to 360. To illustrate this, you can imagine a color circle from 0 to 359 degrees, where the value 0 or 360 stands for red, 120 for green, and 240 for blue.

The values of saturation and lightness are provided in percentages. The more the saturation value is reduced from 100% to 0%, the more the hue turns to gray. The normal lightness, on the other hand, is displayed via the value 50%. A lightness of 100% is white, and a lightness of 0% is black.

Here's a simple example of `hsl()`:

```
.p_article {
  background-color: hsl(240, 100%, 50%);
  color: hsl(0,100%,100%);
}
```

As with the other examples, you use the class selector to define a CSS rule for a paragraph text with the `<p>` element with a blue background and white font.

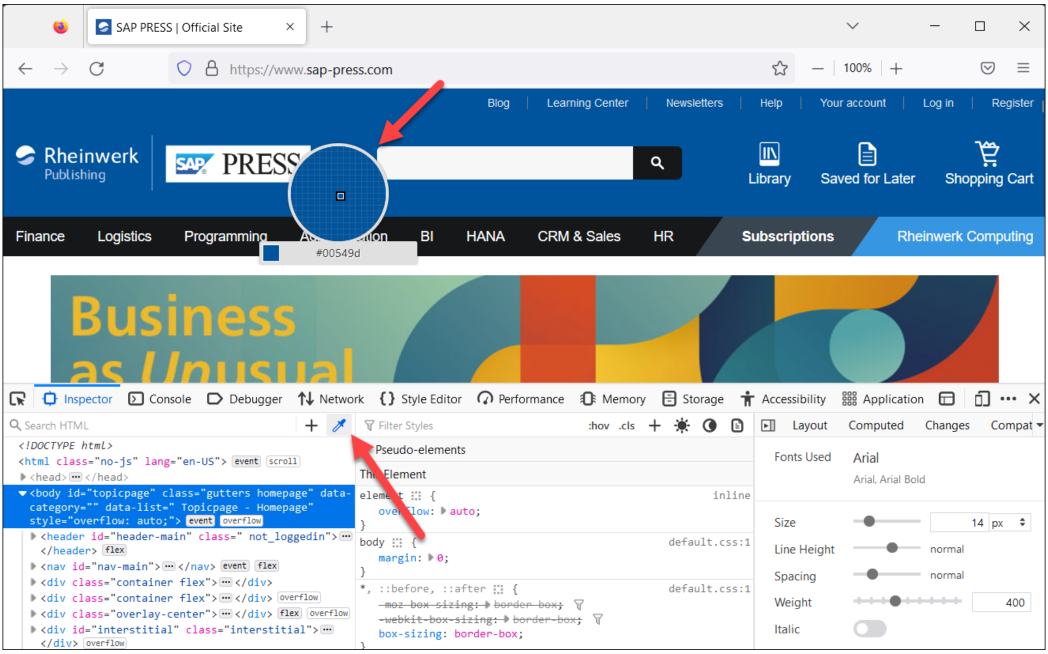
HSL Mixture with Transparency

As with the RGB mixtures with the `rgba()` function, there's an HSL mixture with `hsla (hue, saturation, lightness, opacity)`, where the opacity can be specified as a fourth parameter with a floating-point number from `0` (completely transparent) to `1` (no transparency). The *a* in hsla stands for the alpha channel (opacity).

Color Selection Tools

If you want to know a specific color value on a website, Firefox provides a color picker in the developer tools. If you select it and move the cursor to the respective color, you'll know the color value. If you click on it, the hex value will be copied to the clipboard.

For Google Chrome, you can use the *ColorZilla* extension, and for Microsoft Edge I recommend *ColorFish Color Picker*. The two extensions provide more functions than just selecting the color. ColorZilla is also available for Firefox.



SAP PRESS | Official Site
https://www.sap-press.com
100%
Blog
Learning Center
Newsletters
Help
Your account
Log in
Register
Rheinwerk Publishing
SAP PRESS
Library
Saved for Later
Shopping Cart
Finance
Logistics
Programming
BI
HANA
CRM & Sales
HR
Subscriptions
Rheinwerk Computing
#00549d
Business as Unusual
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Search HTML
Filter Styles
:hov .cls
Pseudo-elements
This Element
Layout
Computed
Changes
Compat

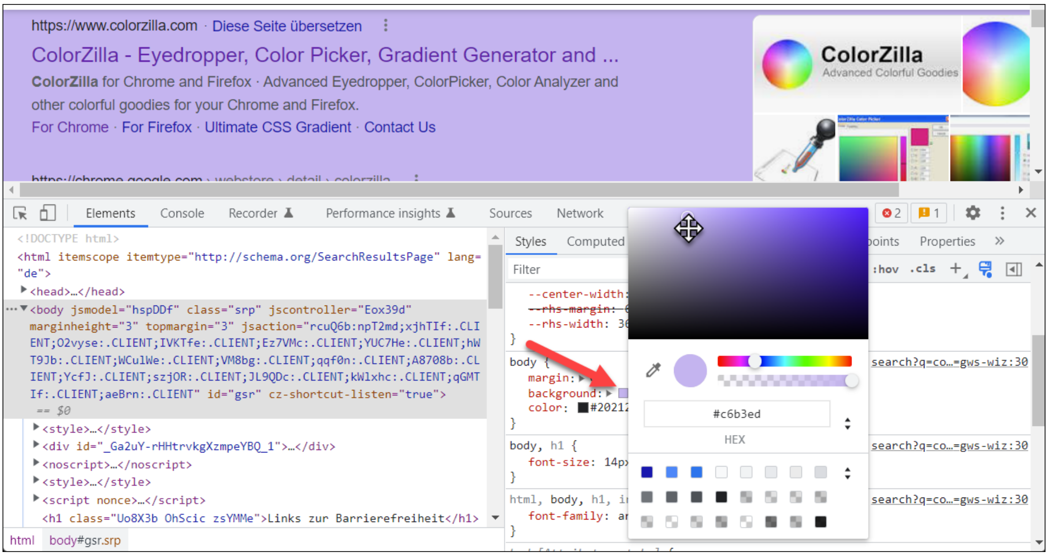
```
<!DOCTYPE html>
<html class="no-js" lang="en-US"> event scroll
 <head>…</head>
 <body id="topicpage" class="gutters homepage" data-
 category="" data-list=" Topicpage - Homepage"
 style="overflow: auto;"> event overflow
  <header id="header-main" class=" not_loggedin">…
  </header> flex
  <nav id="nav-main">…</nav> event flex
  <div class="container flex">…</div>
  <div class="container flex">…</div> overflow
  <div class="overlay-center">…</div> flex overflow
  <div id="interstitial" class="interstitial">…
  </div> overflow
```

```
element { inline
  overflow: auto;
}
body { default.css:1
  margin: 0;
}
*, ::before, ::after { default.css:1
  ~~-moz-box-sizing: border-box;~~
  ~~-webkit-box-sizing: border-box;~~
  box-sizing: border-box;
```

Fonts Used	Arial Arial, Arial Bold
Size	14 px
Line Height	normal
Spacing	normal
Weight	400
Italic	

Figure 10.12 Firefox Color Picker

In the Chrome, Edge, and Firefox web browsers, there's also another useful helper related to the color and CSS.



https://www.colorzilla.com · Diese Seite übersetzen

ColorZilla - Eyedropper, Color Picker, Gradient Generator and ...

ColorZilla for Chrome and Firefox · Advanced Eyedropper, ColorPicker, Color Analyzer and other colorful goodies for your Chrome and Firefox.

For Chrome · For Firefox · Ultimate CSS Gradient · Contact Us

ColorZilla
Advanced Colorful Goodies
Elements
Console
Recorder
Performance insights
Sources
Network
Styles
Computed
Filter
points
Properties
:hov .cls
#c6b3ed
HEX
html
body#gsr.srp

```
<!DOCTYPE html>
<html itemscope itemtype="http://schema.org/SearchResultsPage" lang=
"de">
 <head>…</head>
 <body jsmodel="hspDDf" class="srp" jscontroller="Eox39d"
 marginheight="3" topmargin="3" jsaction="rcuQ6b:npT2md;xjhTIf:.CLI
 ENT;O2vyse:.CLIENT;IVKTfe:.CLIENT;Ez7VMc:.CLIENT;YUC7He:.CLIENT;hW
 T9Jb:.CLIENT;WCulWe:.CLIENT;VM8bg:.CLIENT;qqf0n:.CLIENT;A8708b:.CL
 IENT;YcfJ:.CLIENT;szjOR:.CLIENT;JL9QDc:.CLIENT;kWlxhc:.CLIENT;qGMT
 If:.CLIENT;aeBrn:.CLIENT" id="gsr" cz-shortcut-listen="true">
  == $0
  <style>…</style>
  <div id="_Ga2uY-rHHtrvkgXzmpeYBQ_1">…</div>
  <noscript>…</noscript>
  <style>…</style>
  <script nonce>…</script>
  <h1 class="Uo8X3b OhScic zsYMMe">Links zur Barrierefreiheit</h1>
```

```
  --center-width:
  ~~--rhs-margin:~~
  --rhs-width: 36
}
body {
  margin:
  background:
  color: #20212
}
body, h1 {
  font-size: 14px
}
html, body, h1, i
  font-family: ar
}
```

```
search?q=co…=gws-wiz:30
search?q=co…=gws-wiz:30
search?q=co…=gws-wiz:30
```

Figure 10.13 Developer Tools: Color Picker Where You Can Change the Colors of Individual HTML Elements

Open the developer tools via `Ctrl`+`Shift`+`I`, for example. Then select the HTML element for which you want to see the stylesheet with the color. In the color details of the stylesheet, you'll find a color plate next to the color value. When you move over it with the mouse pointer, you can switch the different notations by holding the `Shift` key (hex value, color name if available, `hsl() value` or `rgb() value`). If, on the other hand, you click on the color plate, a color selection opens in which you can directly change the color value and also determine it. This allows you to experiment with the colors of a web page. The changes are only temporary.

10.3.4 Angular Dimensions in CSS

CSS also defines some *angular dimensions* that you can use to write rotations. If you use a negative value for these units, which are listed in Table 10.6, the rotation is counterclockwise, converting the negative value to its positive counterpart (`-40deg` would become `320deg`). Positive values are rotated clockwise.

UoM	CSS Name	Description
Degree	`deg`	Angle in *degrees*; a complete circle corresponds to 360 degrees, for example: `transform: rotate(90deg);`
Gradian	`grad`	Angle in gradian (*grad*), a complete circle corresponds to 400 grad. 100 grad corresponds to a 90° turn, for example: `transform: rotate(100grad);`
Radian	`rad`	Angle in *radian*; a complete circle here corresponds to 2 pi = 6.2831853, for example: `transform: rotate(5.5rad);`
Round angle	`turn`	A round angle is a circle rotation (`0.25turn` corresponds to a 90° rotation; `1 turn` = 360°). At the time this book went to press, only Firefox was capable of working with this unit. Example: `transform: rotate(0.25turn);`

Table 10.6 Different Angular Dimensions in CSS

Time Data

You can also find two *time specifications* in the CSS specification. The identifiers `s` for second and `ms` for millisecond are available for this purpose. For example:

```
transition-delay: 2s;
```

10.3.5 Passing Values via Short Notation to a CSS Feature

You can use short notations in CSS to bundle some CSS features and write them down in one go. Thus, take a specification such as the following:

```
.my_article { margin: 25px; }
```

It is merely a short version of the following:

```
.my_article {
  margin-top: 25px;     /* Top margin   */
  margin-right: 25px;   /* Right margin */
  margin-bottom: 25px;  /* Bottom margin */
  margin-left: 25px;    /* Left margin  */
}
```

The CSS feature `margin` is used for an outer margin or distance between the current element and its parent or adjacent element. We'll return to that later.

You can use this short notation not only for more than just four of the same properties but also for specifying four different values in a shorter way:

```
.my_article { margin: 25px 10px 20px 5px; }
```

This shorter specification is identical to the following:

```
.my_article {
  margin-top: 25px;     /* Top margin   */
  margin-right: 10px;   /* Right margin */
  margin-bottom: 20px;  /* Bottom margin */
  margin-left: 5px;     /* Left margin  */
}
```

The only important thing about the short notation is that you know about the clockwise order. Starting from 12 o'clock with `top`, the order in the clock hand is `top`, `right`, `bottom`, and `left`.

If the values of `left` and `right` are the same, you can use the following:

```
article { margin: 25px 10px 20px; }
```

Because the fourth value (`left`) has been omitted, the value of `right` gets used instead, making the value of the left and right margins 10 pixels. You can do the same with `top` and `bottom`:

```
.my_article { margin: 20px 10px; }
```

Here, I've specified only two values, so that 20 pixels apply to `top` and `bottom` and 10 pixels to `right` and `left`. You can also subsequently use the `top`, `bottom`, `left`, or `right` versions to override the value of the general CSS feature for all four sides, for example:

```
.my_article {
  margin: 25px;       /* All four side 25 pixels margin */
  margin-left: 0px;   /* Set left margin to 0 pixels */
}
```

Besides `margin`, there are other CSS features that are short notations of several CSS features. I'll describe the individual CSS features in the following chapters, but want to present a few examples already at this point. Without going into detail about the individual functions, the following lines all contain short notations of multiple CSS features:

```
padding: 20px 30px 10px 20px;
border: 1px solid yellow;
background: orange url('background.jpg') no-repeat left top;
font: bold 16px Arial, Helvetica;
list-style: square inside url(bullet.png);
outline: blue dotted thick;
```

10.4 Summary

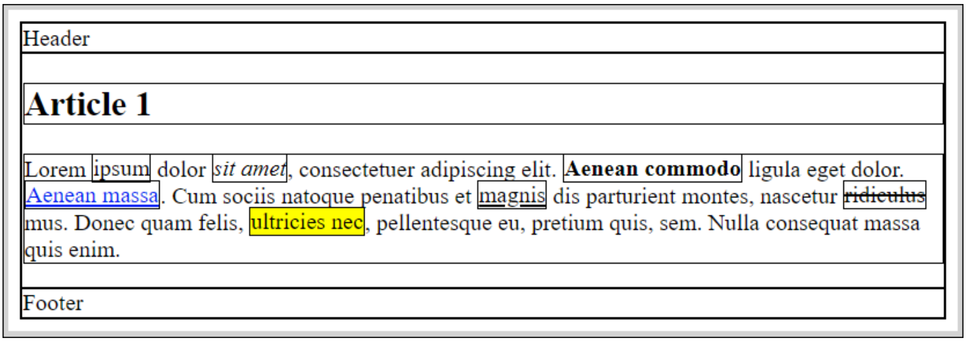
In this chapter, you learned how the principle of inheritance works in CSS and how you can even force such inheritance. Likewise, you now know what is meant by "cascading" in CSS, and thus what CSS does in the event of a conflict, for example, if the same CSS feature has been assigned different values in multiple statements. At the end of the chapter, you learned in a short theoretical digression what kinds of values with and without UoMs you can pass to CSS features.

Chapter 11

The Box Model of CSS

The basis for positioning elements and creating a layout in CSS is the box model. In this chapter, you'll learn about the new and more intuitive box model of CSS in addition to the classic box model.

You've probably noticed that elements on websites consist of rectangular boxes or just boxes. Thanks to those rectangular boxes, it's possible to position elements with CSS or to create the layout of websites.



Header

Article 1

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Footer

Figure 11.1 Websites Consist of Rectangular Boxes (or Just Boxes), Which Were Made Visible with CSS Here

In Figure 11.1, we used the universal selector to make all the rectangular boxes of the HTML elements visible as follows:

```
...
* { border: 1px solid black; }
...
```

Listing 11.1 /examples/chapter011/11/index.html

In this chapter, you'll learn everything you need to know about the following:

- Classic box model of CSS
- Newer alternate box model of CSS
- Styling of boxes with CSS

11.1 Classic Box Model of CSS

In Figure 11.1, you can see only a simple frame so that it seems that such a rectangular box consists only of a height and a width. But the box model also includes the following:

- Actual content area (also content) of text and images, which is specified by `width` and `height`
- Internal spacing (padding), which is specified with `padding`
- Frame, which is written using `border`
- Outer margin, which is specified using `margin`

Thus, a box is composed of four boxes: a *content box* for the content, a *padding box* for the padding, a *border box* for the border, and a *margin box* for the margin. Each of these four boxes, in turn, can be divided into `top`, `right`, `bottom`, and `left`. Figure 11.2 shows a simple representation of the CSS box model.

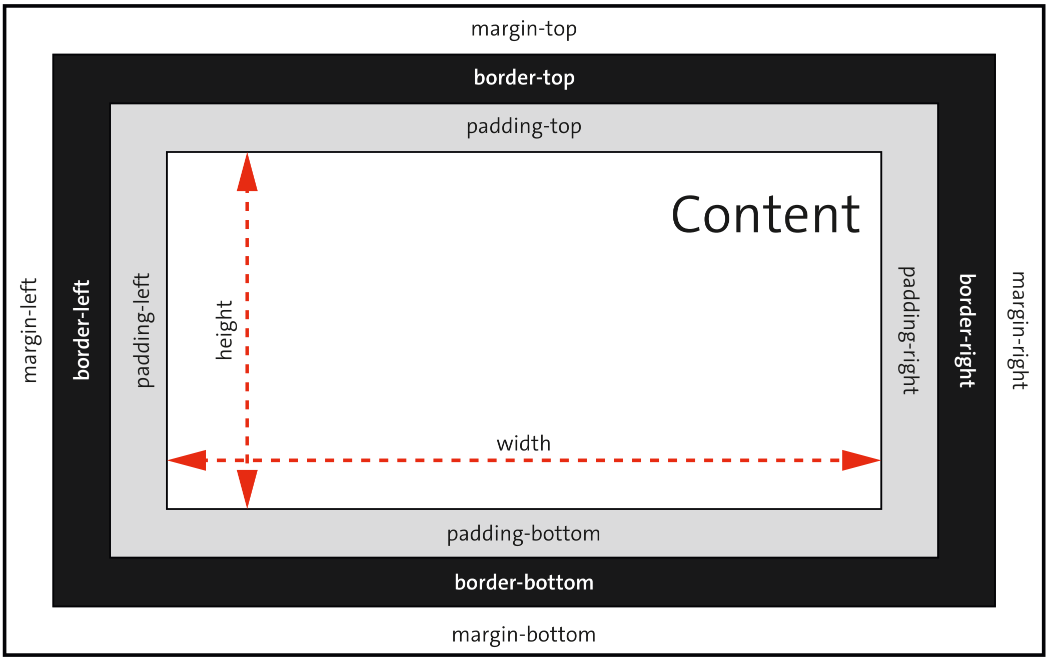

margin-top
border-top
padding-top
Content
margin-left
border-left
padding-left
height
padding-right
border-right
margin-right
width
padding-bottom
border-bottom
margin-bottom

Figure 11.2 Classic CSS Box Model

11.1.1 Specifying the Content Area Using “width” and “height”

You can define the actual content area with the space for text and images using the CSS features `width` and `height` (see also Figure 11.2). If you don’t specify a special value for `width`, the HTML element will be as wide as the surrounding element. If no `height` is specified, all elements will be as high as the content is.

HTML Elements without “width” and “height”

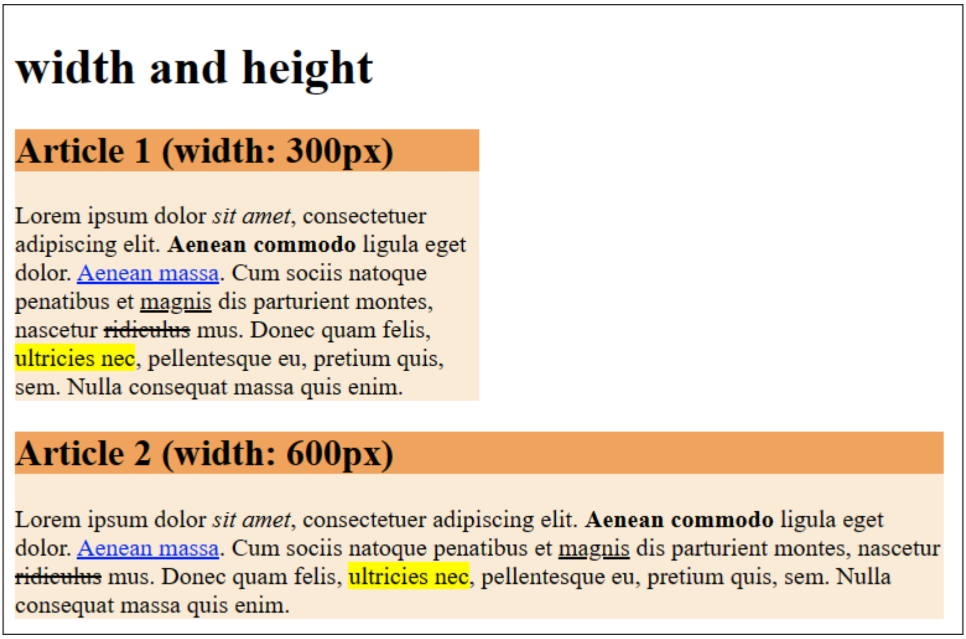
There are elements, for example, `strong`, `abbr`, `a`, or `em`, where the height and width are automatically determined by the extent of the content. You can’t assign the CSS features `width` and `height` to such elements.

For example, in Figure 11.3, two `article` elements were formatted using the following CSS rules:

```
.article_01 { width: 300px; background: antiquewhite; }
.article_02 { width: 600px; background: antiquewhite; }
h_2 { background: sandybrown; }
```

Listing 11.2 /examples/chapter011/11_1/css/style.css

In Figure 11.3, you can see the example */examples/chapter011/11_1_1/index.html* with the CSS code */examples/chapter011/11_1_1/css/style.css* in use.



width and height

Article 1 (width: 300px)

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

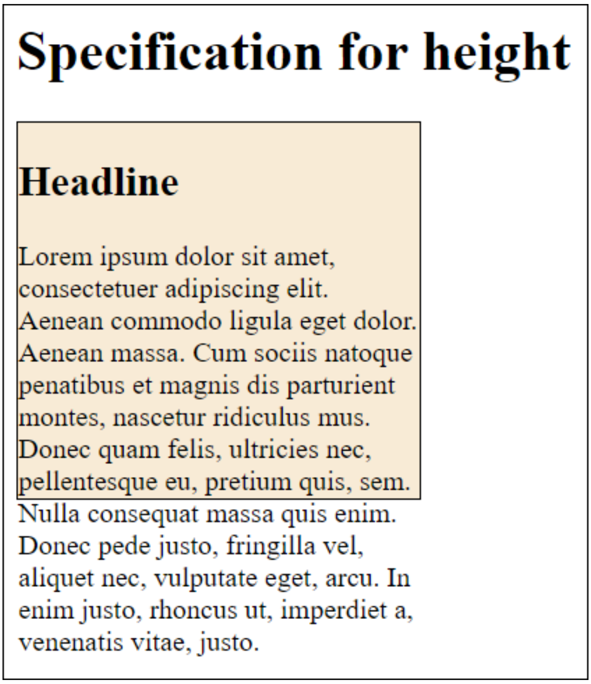
Article 2 (width: 600px)

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Figure 11.3 Two <article> Elements Were Each Defined via a Class Selector with a Fixed Width (“width”): Top Article Is 300 Pixels Wide, and Bottom Article Is 600 Pixels Wide

At this point, it’s slightly confusing at first that specifying the width in the classic box model with the CSS feature `width` doesn’t define the actual width of the element, but only the area with the content (`width` × `height`), as you can see in Figure 11.2.

You should also note that the `height` specification for height is only an initial value. If the content of the encompassing element is larger than the specified height, the content is still displayed and overflows the box, as shown in Figure 11.4. You can find the example for this in */examples/chapter011/11_1_1/index2.html*.



Specification for height

Headline

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo.

Figure 11.4 Text No Longer Fits into the Dimensions Specified for “width” (230 Pixels) and “height” (200 Pixels), Resulting in the Text “Flowing Out” of This Box

In real life, you’ll rarely define fixed values for the width (with `width`) and the height (with `height`). Responsive web design tends to use properties such as `min-width` (minimum width), `min-height` (minimum height), or `max-height` (maximum height) to allow flexible limits to suit the device or screen width.

CSS Feature “overflow”

If you want to prevent overflowing, you could use the CSS feature `overflow` (e.g., with the value `hidden`), which, however, will no longer display the oversized content.

11.1.2 Specifying the Inner Spacing Using “padding”

The inner padding (often only referred to as padding) of the box between the contents and the frame is indicated by `padding` (see Figure 11.2). This CSS feature assumes the background color of the content area if you’ve used one there. The CSS feature `padding` alone sets all four sides clockwise and is a short notation for `padding-top` (top inner spacing), `padding-right` (right inner spacing), `padding-bottom` (bottom inner spacing), and `padding-left` (left inner spacing).

11.1.3 Creating the Border Using "border"

The `border` encloses the `padding` and has its own CSS features for thickness (`width`), line style (`style`), and color (`color`). Similar to padding, you can use `border` to address all four sides at once. Again, you can access each of the four sides separately with `border-top` (top border line), `border-right` (right border line), `border-bottom` (bottom border line), and `border-left` (left border line).

Here's another example, where you mark up two `article` elements, each 600 pixels wide, with `width`. Both `article` elements also get a border with a thickness of 10 pixels. To see the effects of `padding` here, the second `article` element was given an internal spacing (padding) of 50 pixels. Here are the corresponding CSS rules for this:

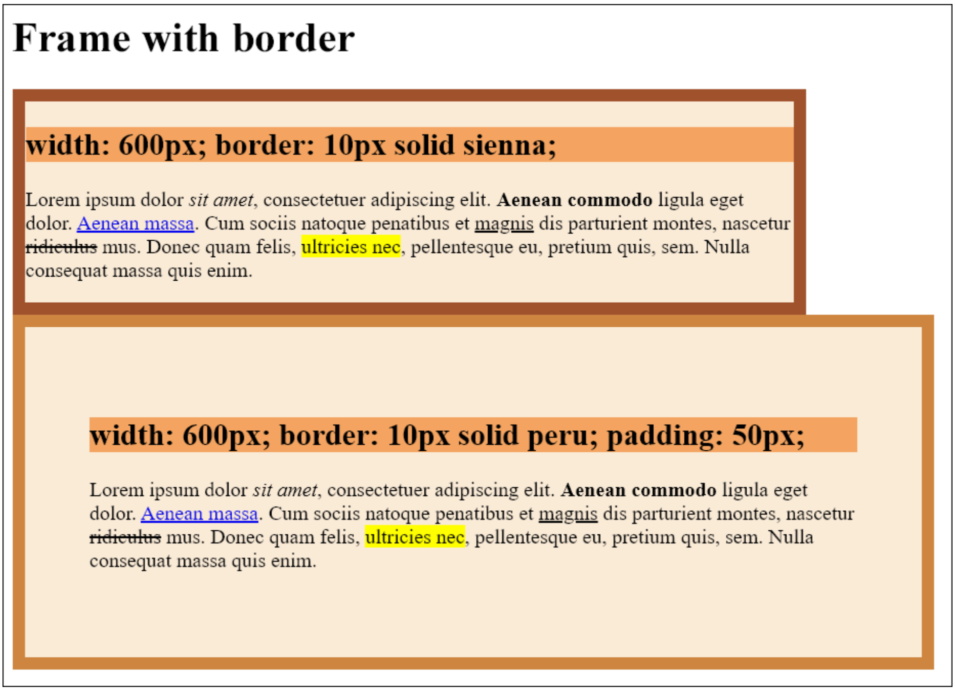
```css
.article01 {
  width: 600px;
  border: 10px solid sienna;
  background-color: antiquewhite;
}
.article02 {
  width: 600px;
  padding: 50px;
  border: 10px solid peru;
  background-color: antiquewhite;
}
.h_2 { background-color: sandybrown;}
```

Listing 11.3 /examples/chapter011/11_1_3/css/style.css

In Figure 11.5, you can see the example */examples/chapter011/11_1_3/index.html* with */examples/chapter011/11_1_3/css/style.css* in use. Both `article` elements have a `width` of 600 pixels and a `border` of 10 pixels. For the bottom `article` element, you've also set up a padding of 50 pixels in all four directions via `padding`. Here, you can see that the padding also takes on the background color of the content area.

11.1.4 Setting Up the Outer Margin Using "margin"

At the very outside of the box model in Figure 11.2 you can see another outer margin with `margin`, which can also be written with `margin-top` (top outer margin), `margin-right` (right outer margin), `margin-bottom` (bottom outer margin), and `margin-left` (left outer margin) for all four directions individually. The outer margin has no color, is completely transparent, and therefore takes on the background color of the surrounding element.



Frame with border

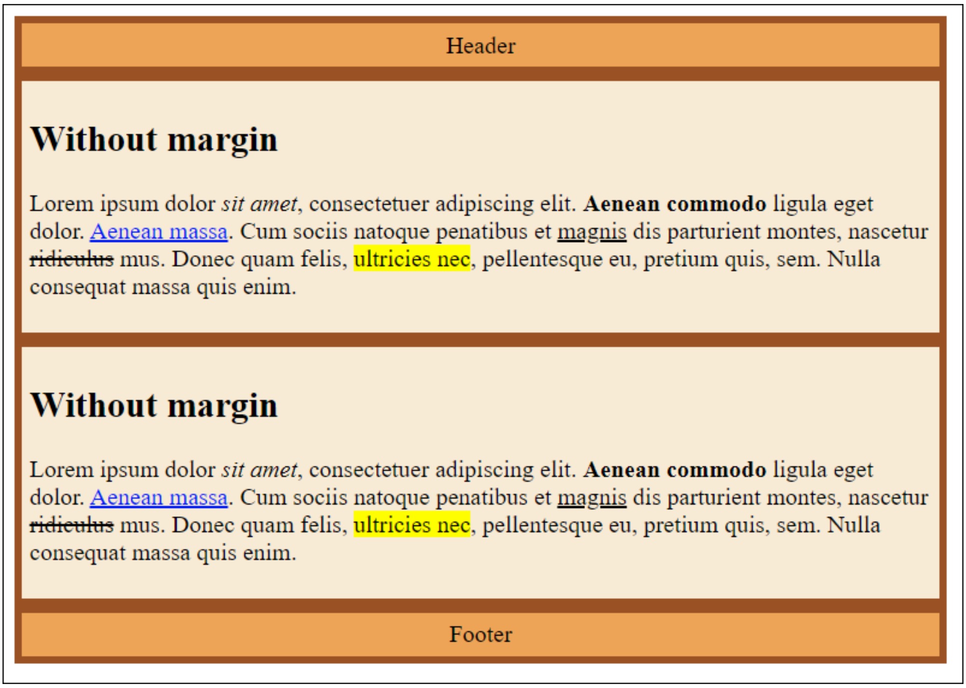
width: 600px; border: 10px solid sienna;

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

width: 600px; border: 10px solid peru; padding: 50px;

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Figure 11.5 A Frame with “border”: One Frame with “padding” and One Without



Header

Without margin

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

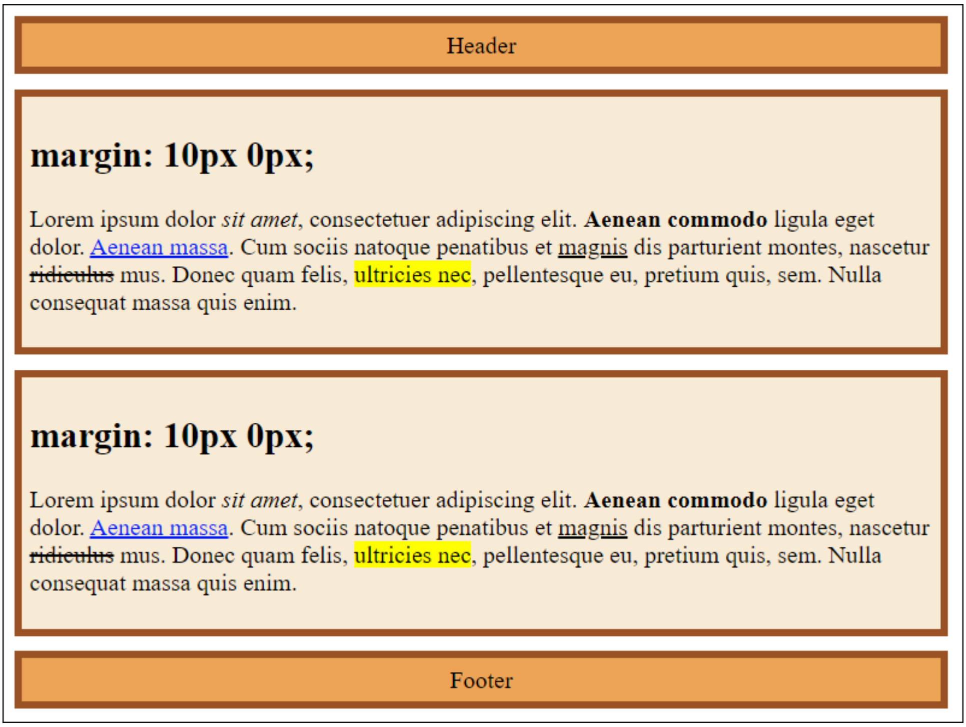
Without margin

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Footer

Figure 11.6 A <header> Element, Two <article> Elements, and One <footer> Element Are Framed, While No Outer Space with “margin” Has Been Used Yet

The example */examples/chapter011/11_1_4/css/style.css* from Figure 11.7 can be found in */examples/chapter011/11_1_4/index.html*. It should also be mentioned here that negative values are allowed for `margin`. The question as to how negative values affect `margin` depends on whether the elements are static, positioned, or floated.



Header

margin: 10px 0px;

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

margin: 10px 0px;

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Footer

Figure 11.7 The <article> Element Has Been Set with an Outer Margin of 10 Pixels to the Top and Bottom (“margin: 10px 0px”)

11.1.5 Collapsing Margins

One initially somewhat confusing peculiarity of vertical margins must definitely be mentioned at this point. It concerns the vertical margins between two boxes placed one above the other. If the two margins touch each other, they won’t be added up, as might be assumed, but only the larger of the two margins will be used. The smaller margin gets virtually “swallowed” by the larger one.

Figure 11.8 demonstrates this process a bit more clearly. Here, two vertical boxes are on top of each other on the *left-hand side*, while the two margins are touching each other. For the top box, a `margin` of 20 pixels to the bottom was specified with `margin-bottom`. In the lower box, on the other hand, a `margin` of 10 pixels to the top was specified with `margin-top`.

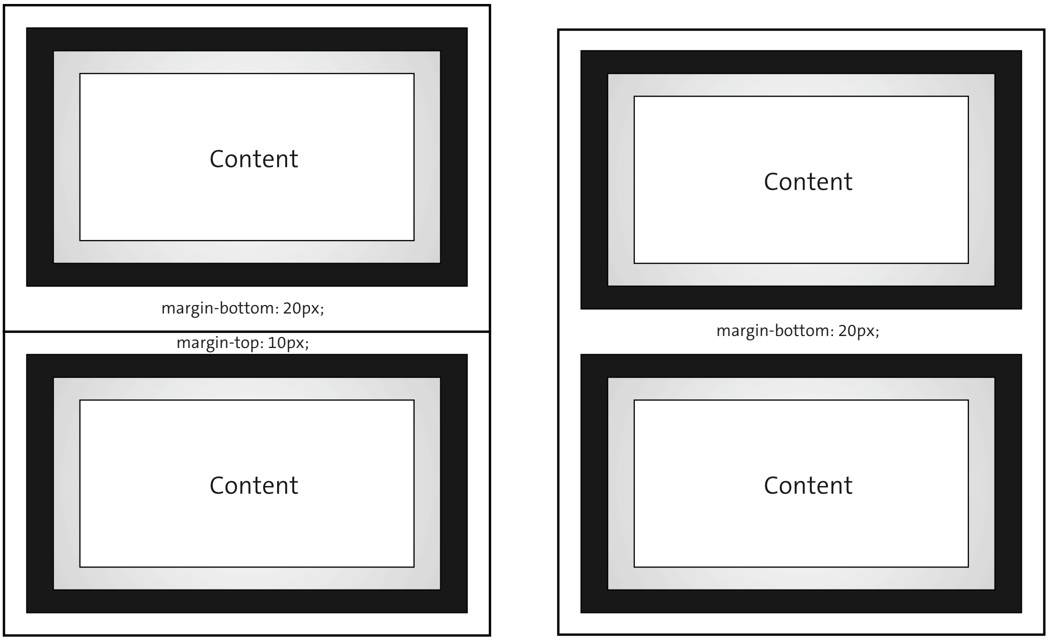

Content
margin-bottom: 20px;
margin-top: 10px;
Content
Content
margin-bottom: 20px;
Content

Figure 11.8 Vertical Margins That Touch Each Other Collapse into the Larger Value of the Two Margins

You can see the result of these two boxes stacked on top of each other on the right, where the two boxes collapse and the smaller of the two margins has been removed.

Nothing would change if you set the `margin-top` value of the bottom box to 19 pixels. Only if you set `margin-top` to 21 pixels would the lower margin be swallowed because with 20 pixels, it would then be the smaller margin.

Here's a simple example that demonstrates how `margin-bottom` and `margin-top` collapse:

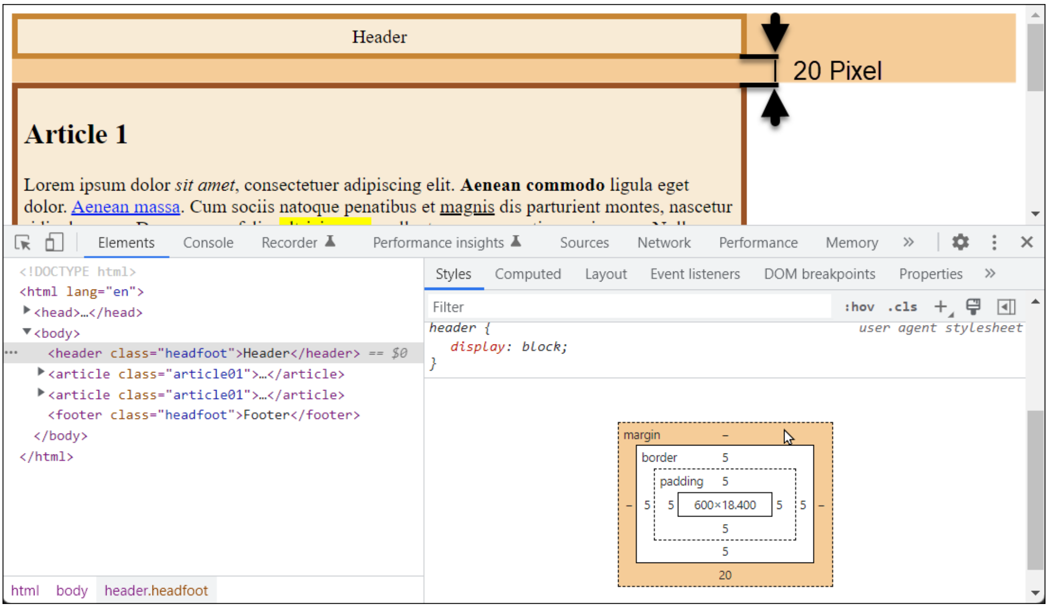
```
.headfoot {
  width: 600px;
  padding: 5px;
  border: 5px solid peru;
  background-color: antiquewhite;
  margin-bottom: 20px;
  text-align: center;
}
.article01 {
  width: 600px;
  padding: 5px;
  border: 5px solid sienna;
  background-color: antiquewhite;
  margin: 10px 0px;
}
```

Listing 11.4 /examples/chapter011/11_1_5/css/style.css

```
...
    <header class="headfoot">Header</header>
    <article class="article01">
       <h1>Article 1</h1>
       <p> ... </p>
    </article>
...
```

Listing 11.5 /examples/chapter011/11_1_5/index.html

In this example, the `header` element, which you address via the `.headfoot` class selector, and the `article` element, which is styled using the `.article01` class selector, are on top of each other. For the `header` element, the `margin` is 20 pixels to the bottom (`margin-bottom: 20px`). For the `article` element, on the other hand, the margin to the top is 10 pixels (`margin: 10px 0px`). In total, the margin between the `header` and `article` elements is 20 pixels (rather than 30 pixels) because of the collapsing margin, which is the larger of the two values. The margin to the top of the `article` element is completely omitted here (it's swallowed, as mentioned earlier), as you can see in Figure 11.9.



Header

Article 1

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <header class="headfoot">Header</header> == $0
   ▶<article class="article01">…</article>
   ▶<article class="article01">…</article>
    <footer class="headfoot">Footer</footer>
  </body>
</html>
```

```
header {                                  user agent stylesheet
  display: block;
}
```

20 Pixel
Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory
Styles
Computed
Layout
Event listeners
DOM breakpoints
Properties
Filter
:hov
.cls
margin –
border 5
padding 5
600×18.400
5
20
html
body
header.headfoot

Figure 11.9 Two Collapsing Margins: Instead of the Mathematically Logical 30 Pixels, the Distance Here Is 20 Pixels

Horizontal Margins Don't Collapse

When boxes touch each other horizontally, the margins don't collapse but are added up normally.

The collapsing is intentional and serves to keep the spacing even for texts that consist of multiple paragraphs. For example, you can format a paragraph text with the following CSS rule:

```
p { margin: 1.2em; }
```

You then can obtain the following HTML structure, for example:

```
<h1>...</h1>
<p>1stParagraph</p>
<p>2ndParagraph</p>
<p>3rdParagraph</p>
```

In this way, you can be sure that the spacing of all three paragraphs is `1.2em` thanks to the collapsing margins. If the margins didn't collapse, the margin of `<h1>` to the first `p` element would still be `1.2em`, but the margin of the second `p` element to the first and to the third `p` element would be `2.4em`. Such giant gaps wouldn't look nice at all.

So, when two vertical boxes touch each other on the outside, the following rules apply when they collapse or even merge:

- If both values are the same, only one of them will be taken over.
- If the values differ, the larger value will be used.

Preventing Collapsing Margins

The nasty thing about these collapsing margins is that you don't even notice them as a problem at first, when the layout doesn't want to work that way. Especially for headings, paragraphs, lists, or quotes, you can counteract this a bit by setting `margin-top` for these elements to `0`. You can now easily control the distance to the next element with `margin-bottom`. In this way, the collapse almost doesn't occur. The CSS rule for this looks as follows:

```
h1, h2, h3, h4, h5, h6,
p, ul, ol, blockquote {
   margin-top: 0;
}
```

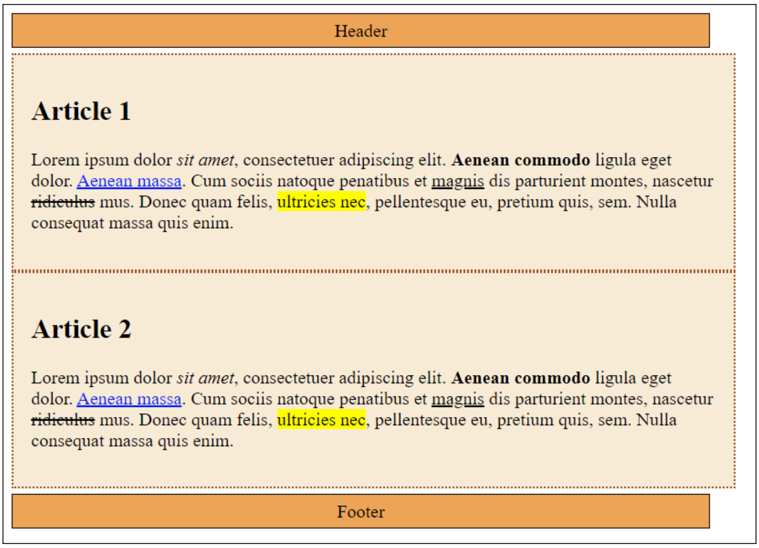
11.1.6 Determining the Total Width and Total Height of a Box

Now that you know all the components of the classic box model, you can calculate the total width or height of a box. As an example, consider the following CSS code:

```
.headfoot {
  width: 600px;
  padding: 5px;
  border: 1px solid black;
  background-color: sandybrown;
  margin: 5px 0px;
  text-align: center;
}
.article01 {
  width: 600px;
  padding: 15px;
  border: 2px dotted sienna;
  background-color: antiquewhite;
}
```

Listing 11.6 /examples/chapter011/11_1_6/css/style.css

I've already pointed out that the `width` specification can be somewhat confusing because `width` doesn't correspond to the actual `width` of a box. As you can clearly see in Figure 11.10 of */examples/chapter011/11_1_6/index.html*, the header and footer are each shorter than the two articles, although in the example for all elements in */examples/chapter011/11_1_6/css/style.css*, the specified width is 600 pixels (`width: 600px`).



Header

Article 1

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Article 2

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Footer

Figure 11.10 Despite Identical Width Specifications with "width", the Boxes Are Displayed with Different Widths; To Adjust This Value, You Need to Calculate the Total Width

You can calculate the total width of a box by adding up `width`, `padding-right`, `padding-left`, `border-right-width`, `border-left-width`, `margin-right`, and `margin-left`. With regard to the two class selectors `.headfoot` and `.article01`, you'll obtain the calculation listed in Table 11.1 as a result. This result shows that it can be helpful to do some simple math if the boxes don't fit. In this example, the total width of the `header` and `footer` elements is 612 pixels. In contrast, the total width of the two `article` elements is 634 pixels.

CSS Feature	.headfoot	.article
`width`	600 pixels	600 pixels
+ `padding-right`	5 pixels	15 pixels
+ `padding-left`	5 pixels	15 pixels
+ `border-right-width`	1 pixel	2 pixels
+ `border-left-width`	1 pixel	2 pixels
+ `margin-right`	0	0
+ `margin-left`	0	0
Total width	**612 pixels**	**634 pixels**

Table 11.1 If the Boxes Don't Fit, a Simple Addition Exercise Can Be Useful

Thus, the difference in total width between the `header` and `footer` elements and the `article` element is exactly 22 pixels (634 pixels – 612 pixels). At this point, you need to decide for yourself where to add or remove these 22 pixels. For example, one solution could be to set the `width` value for the `.article01` class selector to `578px`. You can find the CSS example in */examples/chapter011/11_1_6/css/style.css* and the HTML document for it in */examples/chapter011/11_1_6/index2.html*.

11.2 Newer Alternate Box Model of CSS

Not everyone will like the classic box model, where you specify `width` to determine the width of the content area and end up having to consider `padding`, `border`, and `margin` for the total width. However, as long as the specifications are made in terms of pixels, this is still a cumbersome calculation, but you can always realize a neat layout this way.

It gets a bit more complicated if you use different units for `width`, `padding`, `border`, or `margin`. For example, if you've specified a column with a `width` of 30% in a two-column layout and have written 5 pixels each for `padding` and `border`, it will be difficult to tell exactly how much space this column actually takes up in the layout. The problem was solved by using an inner `<div>` inside the corresponding column to specify `padding`, `border`, or `margin` there instead of in the actual column. For the actual column, only the `width` specification was used instead.

To make a long story short, with the newer alternate box model, you don't have this problem with the math or trickery of adding another `<div>` inside a column for `padding`, `border`, or `margin`. With the new box model **`border-box`**, the `width` and `height` are no longer specified "only" for the content area, but these specifications also sensibly take into account the `padding` and the `border`. The CSS features `width` and `height` are the width and height from `border-left` to `border-right` and from `border-top` to `border-bottom`, respectively, as you can see in Figure 11.11.

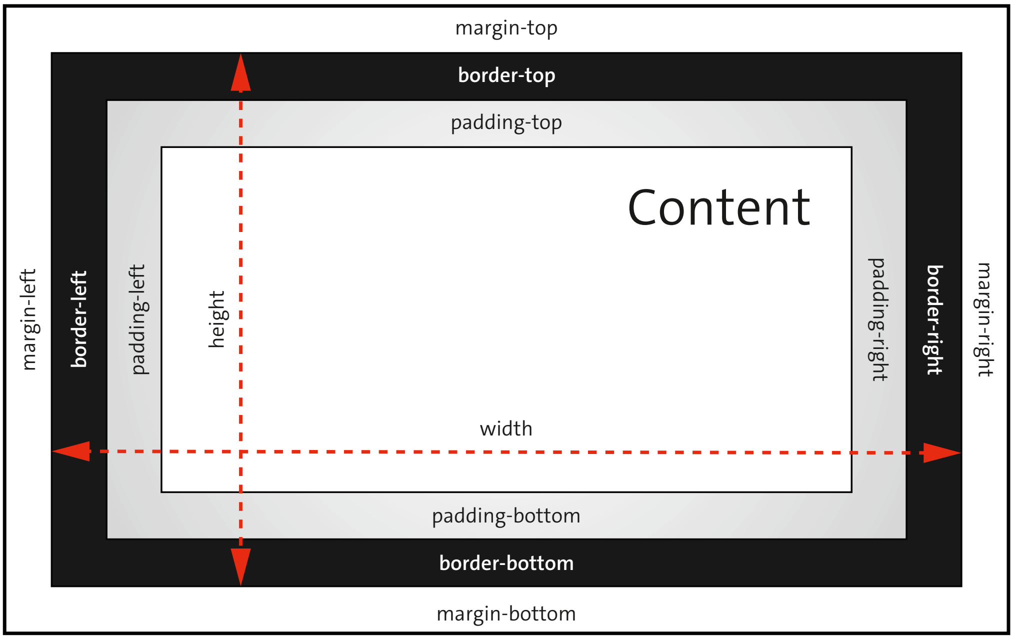

margin-top
border-top
padding-top
Content
margin-left
border-left
padding-left
height
padding-right
border-right
margin-right
width
padding-bottom
border-bottom
margin-bottom

Figure 11.11 The New Box Model "border-box" Makes Your CSS Life a Lot Easier

If you write a width specification with `width` in this box model, `padding` and `border` will no longer have any influence on this specification and are subtracted from this width. Figure 11.12 shows the difference between the classic box model (top left) and the alternate box model (bottom left and top right) in terms of the CSS features `width` and `height`.

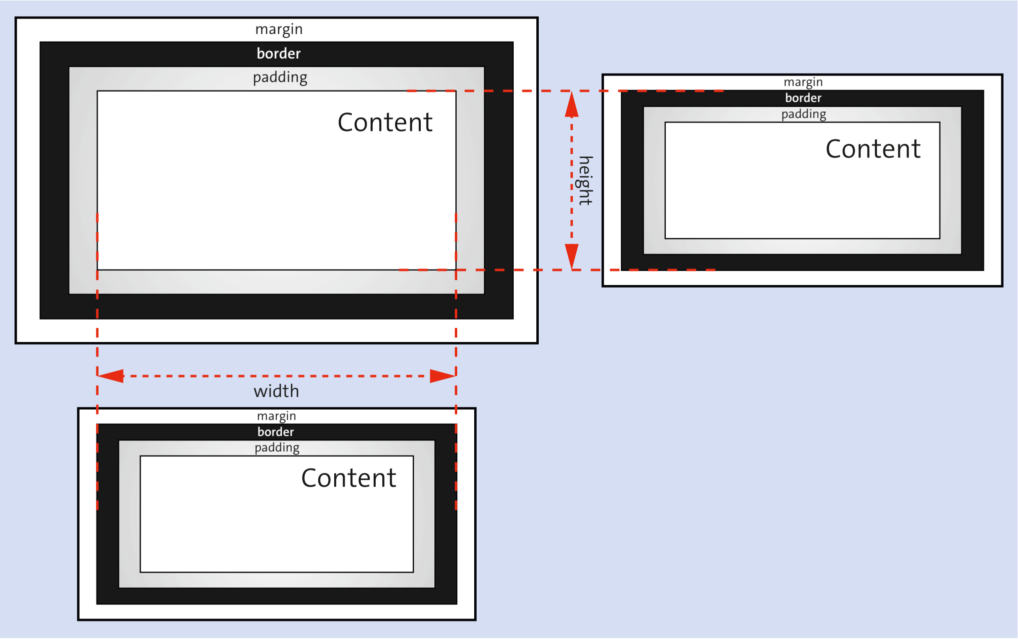


Figure 11.12 Top Left Shows the Classic Box Model; Bottom Left and Top Right Show the New Box Model with "box-sizing" Compared to the Width and Height Specifications

11.2.1 Using the "box-sizing" Box Model

To use the alternate box model, you must assign the `border-box` value to the CSS feature `box-sizing`. Possible values you can use for `box-sizing` are listed here:

- **`content-box`**
  This corresponds to the behavior of the classic box model, where the specification of the `width` and `height` corresponds to the content of the element in the box.
- **`border-box`**
  As already described, in this specification, the value for `width` and `height` corresponds to the value from `border-left` to `border-right` and from `border-top` to `border-bottom`, respectively. Changing `padding` and `border` won't change the width or height of the element anymore. You can see this box model in Figure 11.11 and Figure 11.12.
- **`inherit`**
  This option allows you to adopt the value from the parent element.

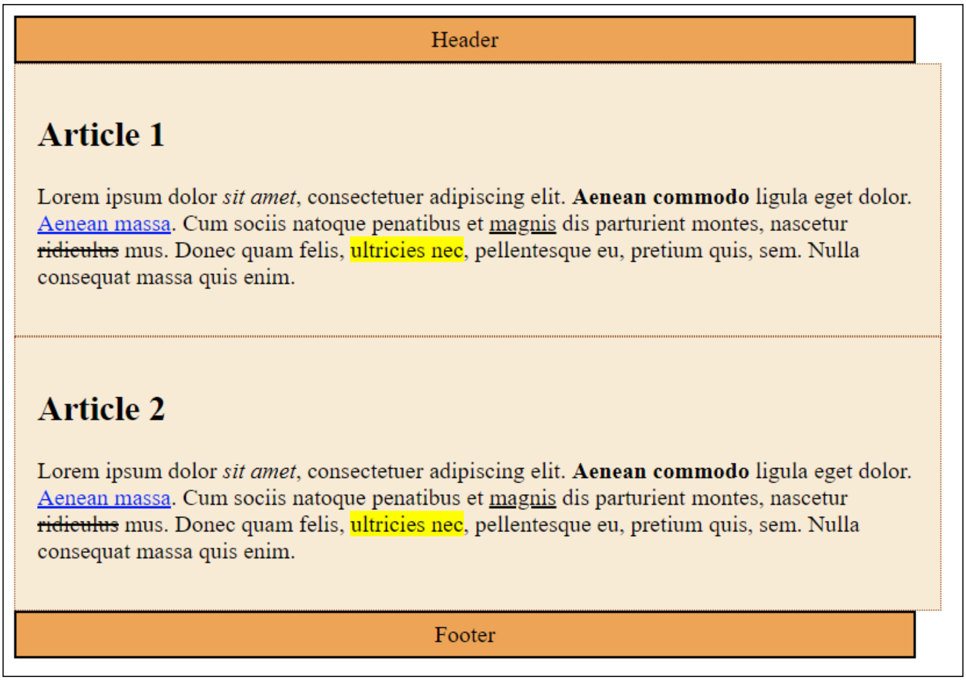
11.2.2 Using the Alternate Box Model

Here's an example to demonstrate the difference in use between the classic model and the alternate box model. Again, a header and footer (`<header>`, `<footer>`) and two articles (`<article>`) are styled, respectively:

```
.headfoot {
  width: 70%;
  padding: 5px;
  border: 2px solid black;
  background-color: sandybrown;
  text-align: center;
}
.article01 {
  width: 70%;
  padding: 15px;
  border: 1px dotted sienna;
  background-color: antiquewhite;
}
```

Listing 11.7 /examples/chapter011/11_2/css/style.css

In Figure 11.13, you can see in */examples/chapter011/11_2_2/index.html* that despite the width specification of 70% for the class selectors `.headfoot` and `.article01` in */examples/chapter011/11_2_2/css/style.css*, the boxes are arranged differently because different specifications were made for `padding` and `border`.



Header

Article 1

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Article 2

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Footer

Figure 11.13 The Inconsistent Representation with the Classic Box Model

The problem with such a layout could be fixed relatively easily by adjusting the values of `padding` and `border`, for example, or even easier, by using the new box model with `border-sizing: border-box;`. Here's the CSS snippet of the example with the new box model:

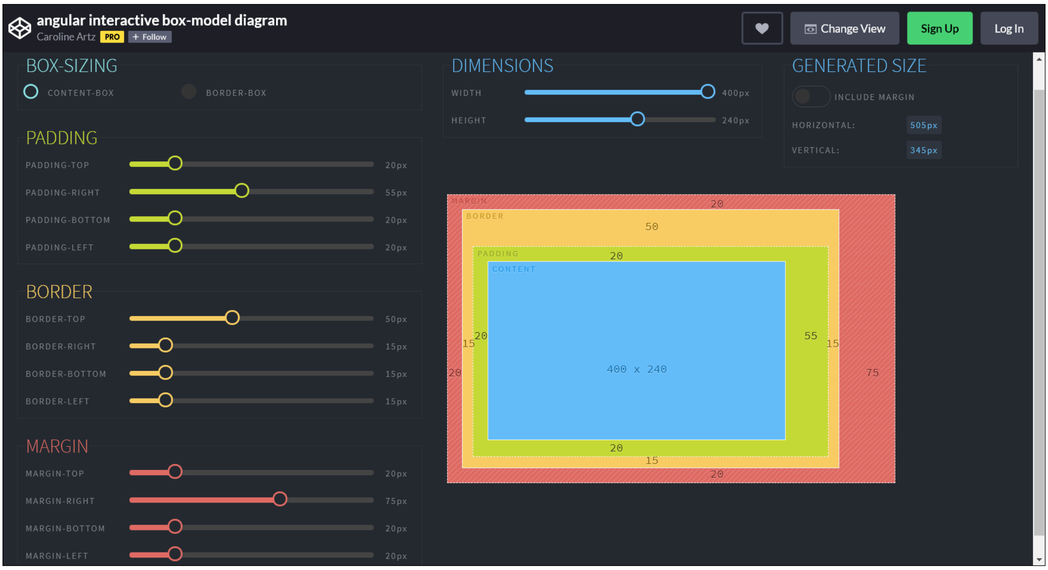
```
.headfoot {
  box-sizing: border-box;
  width: 70%;
  padding: 5px;
  border: 2px solid black;
  background-color: sandybrown;
  text-align: center;
}
.article01 {
  box-sizing: border-box;
  width: 70%;
  padding: 15px;
  border: 1px dotted sienna;
  background-color: antiquewhite;
}
```

You can avoid this double use of the box model specification by putting the new box model into a universal selector right at the beginning of the stylesheet. The alternative to the example just shown would look as follows:

```
* {
  box-sizing: border-box;
}
.headfoot {
  width: 70%;
  padding: 5px;
  border: 2px solid black;
  background-color: sandybrown;
  text-align: center;
}
.article01 {
  width: 70%;
  padding: 15px;
  border: 1px dotted sienna;
  background-color: antiquewhite;
}
```

Listing 11.8 /examples/chapter011/11_2/css/style2.css

As you can see in Figure 11.15 of */examples/chapter011/11_2/index2.html*, all boxes are displayed consistently at 70%, no matter what values you used for `padding` and `border`. By using the new box model with `border-box`, `padding` and `border` get subtracted from the width and are no longer added. Similarly, this applies to the height via `height`.



BOX-SIZING
○ CONTENT-BOX ● BORDER-BOX

PADDING
PADDING-TOP 20px
PADDING-RIGHT 55px
PADDING-BOTTOM 20px
PADDING-LEFT 20px

BORDER
BORDER-TOP 50px
BORDER-RIGHT 15px
BORDER-BOTTOM 15px
BORDER-LEFT 15px

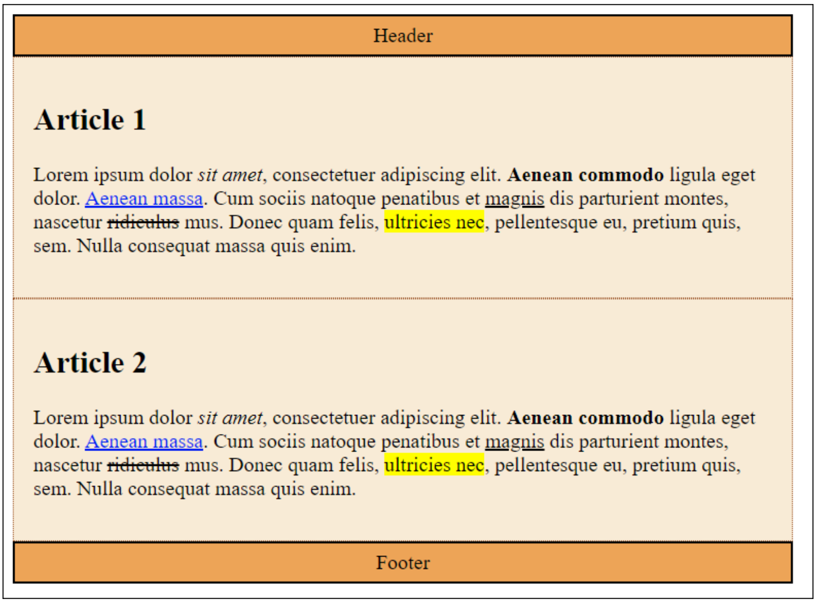
MARGIN
MARGIN-TOP 20px
MARGIN-RIGHT 75px
MARGIN-BOTTOM 20px
MARGIN-LEFT 20px

DIMENSIONS
WIDTH 400px
HEIGHT 240px

GENERATED SIZE
INCLUDE MARGIN
HORIZONTAL: 505px
VERTICAL: 345px

Figure 11.14 An Interactive Box Model

The new box model with `box-sizing` can indeed simplify CSS life considerably. Its strengths come into play when you use percentage values for width and pixel values for `padding` or `border`, for example, when you mix different units. For example, if you've defined a column with 30% `width` with `box-sizing: border-box;`, it doesn't matter what values and units you use for `padding` and `border`—it will remain 30% for the total width of the column.



Header

Article 1

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Article 2

Lorem ipsum dolor *sit amet*, consectetuer adipiscing elit. **Aenean commodo** ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ~~ridiculus~~ mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Footer

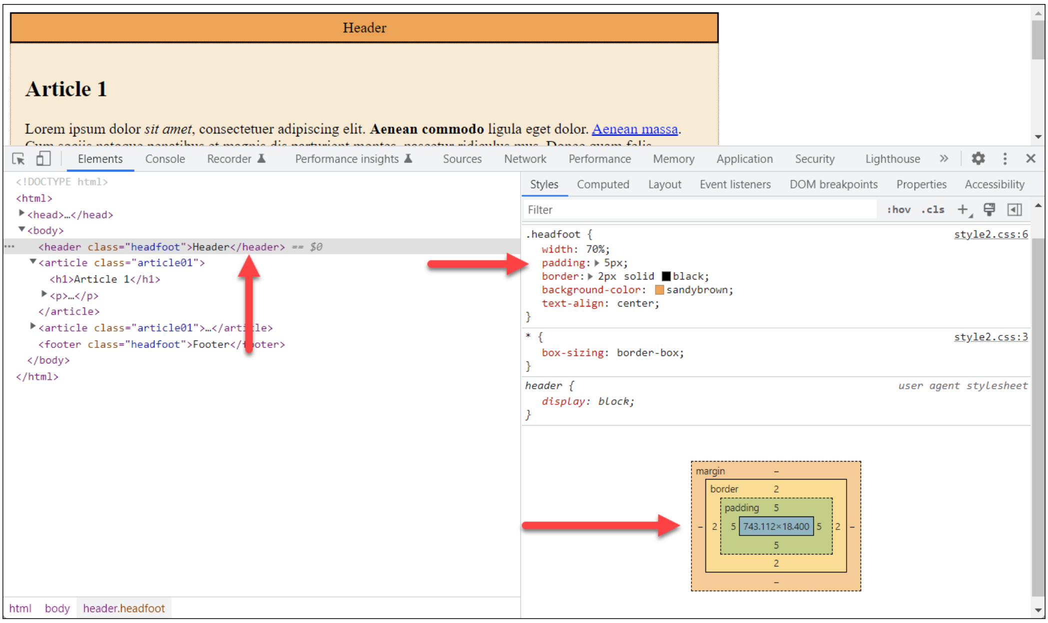
Figure 11.15 The Display with the New Box Model No Longer Causes Any Problems

Interactive Box Model Diagram

A great interactive box model, both with the classic and alternate box model, can be found at *https://codepen.io/carolineartz/full/ogVXZj*. Depending on the selected box model, you can adjust the individual specifications such as `width`, `height`, `padding`, `border`, and `margin` and see how these settings will affect the box model. The interactive model also helps immensely to get a feel for the box model in CSS.

11.3 Analyzing the Box Model in the Browser

In addition to the great interactive box model on the *https://codepen.io/carolineartz/full/ogVXZj* website, web browsers provide appropriate developer tools to visualize and analyze the box model in the browser. If you call the developer tools, which you can do via `Ctrl` + `Shift` + `I` in most web browsers, you'll also find the box model of the selected HTML element, including the pixel values for **margin**, **border**, **padding**, and the content.


Header
Article 1
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.
Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Styles
Computed
Layout
Event listeners
DOM breakpoints
Properties
Accessibility
Filter
:hov
.cls
margin
–
border
2
padding
5
743.112×18.400
html
body
header.headfoot

```
<!DOCTYPE html>
<html>
 ▸<head>…</head>
 ▾<body>
    <header class="headfoot">Header</header> == $0
   ▾<article class="article01">
      <h1>Article 1</h1>
     ▸<p>…</p>
    </article>
   ▸<article class="article01">…</article>
    <footer class="headfoot">Footer</footer>
  </body>
</html>
```

```
.headfoot {                                   style2.css:6
   width: 70%;
   padding: ▸ 5px;
   border: ▸ 2px solid ■black;
   background-color: ■sandybrown;
   text-align: center;
}
* {                                           style2.css:3
   box-sizing: border-box;
}
header {                              user agent stylesheet
   display: block;
}
```

Figure 11.16 Visualizing and Analyzing the Box Model in the Web Browser

If you now move the mouse cursor over the content or one of the **margin**, **border**, or **padding** areas in the visualized box model, this area will be highlighted in the corresponding color of the selected element in the web browser. Here you can determine if the element matches what you intended with the CSS statements, detect errors, or just get to know the box model of CSS. Furthermore, you can use this box model to change the values on a test basis. To do this, you double-click on the corresponding value, change it, and then immediately view this change live in the browser. The changes are made only temporarily in the web browser.

11.4 Box Model for Inline Elements

You already know that everything consists of rectangular boxes in HTML. This also applies to inline elements such as `em`, `strong`, or `a`. There's also a box model available for that, which of course differs from the model described so far. For example, you can't specify a height or width for inline elements, and accordingly, of course, the extended CSS features such as `min-width`, `max-width`, `min-height`, or `max-height` have no effect on inline elements. Rather, the content determines the height and width. You can manipulate the height via `font-size`.

There are also differences in `margin`, `border`, and `padding` for the `-top` and `-bottom` versions. While everything on the right and left can be used as before, `margin-top` and `margin-bottom` have no effect on inline elements. `padding-top` and `padding-bottom` as well as `border-top` and `border-bottom` can be used, but this doesn't change the line height, which is why it can have a negative effect on the readability of the text.

11.5 Designing Boxes

The topic of the box model also includes the styling of boxes. This isn't yet about the layout, but mainly about the visual design such as the background or the frame of boxes. Specifically, you'll learn the following in this section:

- How to design the frame
- How to set the background color
- How to use background images
- How to use transparency
- How to add a gradient
- How to create a shadow
- How to make the square boxes round

11.5.1 Adding and Designing a Border Using the "border" Property

For each element, you can display a border and customize the border color, line width, and type. Even though a border can be used for all elements, the use of a `border` seems to make sense only for HTML elements that create their own paragraph. To understand the use of borders, you should be familiar with the box model, which I covered in detail in Section 11.1 and Section 11.2.

Designing All Four Sides at Once with the Short Notation

Because there are many `border` properties, creating borders is relatively flexible and versatile. In real life, you'll often use the short notation with `border`:

```
/* border: Border color Line width Line style */
border: black 1px solid;
```

This draws a `solid` `black` border with a line width of one pixel (`1px`) around all four sides. You can define this short notation with three CSS features:

```
border-color: black;    /* Border color */
border-width: 1px;      /* Line width */
border-style: solid;    /* Line style   */
```

This specifies the same properties as the short notation of `border` in the previous example.

Four Styles for "border-style" and "border-color"

It's possible to specify two, three, or four values for the two CSS features `border-style` or `border-color`. Specify a value such as the following:

```
border-color: black;
```

This will cause everything to apply as before to all four sides. On the other hand, if you specify two values, such as the following:

```
border-color: red green;
```

This will cause the top and bottom borders to be colored red and the left and right borders to be colored green. If you use three values, the top border is styled with the first value, the right and left borders with the second, and the bottom border with the third. In addition, of course, you can still write all four sides individually:

```
border-color: red green blue yellow;
```

Here, the border colors are assigned clockwise starting with `top`. Similarly, all this also applies to the `border` style with `border-style`.

Designing Each Side of the Border Individually

Similar to how the CSS feature `border` can be used as a bundle for `border-color`, `border-width`, and `border-style` for all four sides, `border-top`, `border-right`, `border-bottom`, and `border-left` each have a CSS feature that you can use to access each side of the border separately. For example, if you want to define only the upper side of a border, you can do it as follows:

```
border-top: red 5px dotted;
```

This adds a `dotted` `red` border with a line width of 5 pixels to the top of the box only. Again, this is ultimately a summarizing short notation of the following:

```
border-top-color: red;
border-top-width: 5px;
border-top-style: dotted;
```

Similarly, there are summarizing properties with `border-right`, `border-bottom`, and `border-left` for each individual side of the border. Needless to say, these summarizing properties are also available in a short notation in the form of `border-...color`, `border-...-width`, and `border-...-style` for the respective sides `right`, `bottom`, and `left`.

Using Different Border Styles

As you've already noticed, there are different styles of borders. You can use `solid` to display a solid border. Table 11.2 contains an overview of the different border styles you can use.

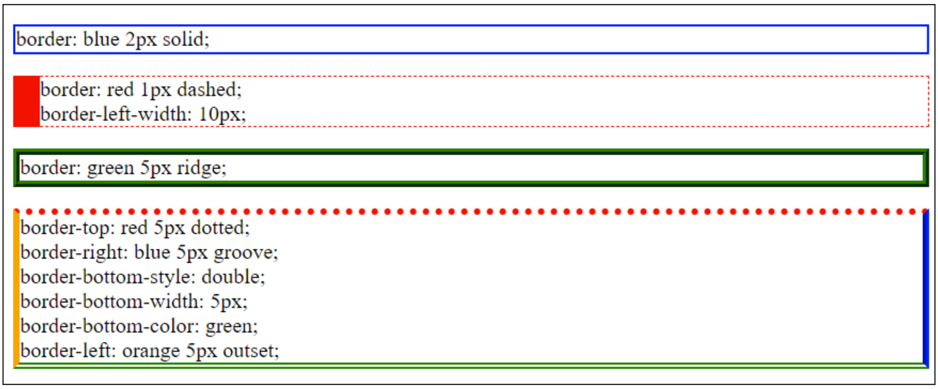
Value	Description
`none`	Default value, no border
`hidden`	Like `none`, no border
`dotted`	Displays a dotted border
`dashed`	Displays a dashed border
`solid`	Displays a solid border
`double`	Displays a double solid border

`groove`
`ridge`
`inset`
`outset`
	Displays a 3D border, the effect also depends on the border color

Table 11.2 Different Border Styles at a Glance

Experiment with the Different Features

Printing a comprehensive example here probably doesn't make much sense as the design of border with the many different `border` properties is extremely diverse. So, I recommend you experiment a bit with the example in */examples/chapter011/11_5_1/css/style.css* to get a feel for it. The corresponding HTML document can be found in */examples/chapter011/11_5_1/index.html*.


border: blue 2px solid;

border: red 1px dashed;
border-left-width: 10px;

border: green 5px ridge;

border-top: red 5px dotted;
border-right: blue 5px groove;
border-bottom-style: double;
border-bottom-width: 5px;
border-bottom-color: green;
border-left: orange 5px outset;

Figure 11.17 Some Different Border Styles in Use (Example in /examples/chapter011/11_5_1/index.html)

Creating a Decorative Border Using "border-image"

`border-image` allows you to easily insert an image for the border. For this purpose, a pixel graphic or Scalable Vector Graphic (SVG) is sufficient, in which the areas for the decorative border are located. Let's take a look at the following 150 × 150 pixel graphic in Figure 11.18 with colored circles of 50 × 50 pixels in each. Because a vector graphic can be scaled without quality restrictions and is also much smaller than a pixel graphic, I used an SVG for this example.

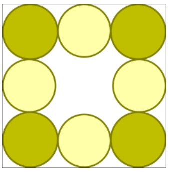


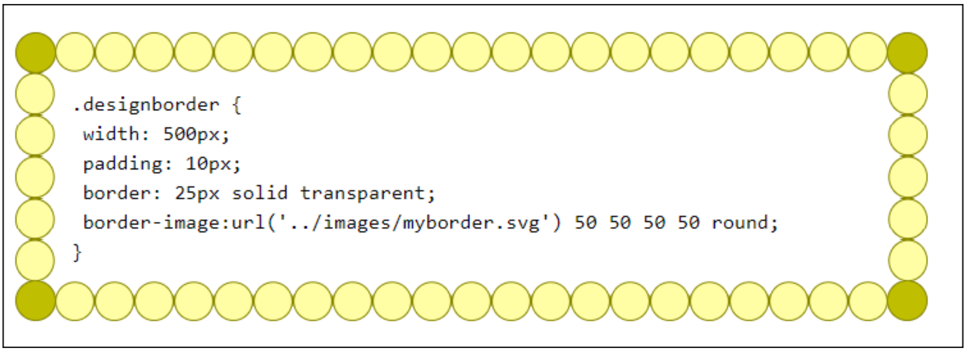
Figure 11.18 The 150 × 150 Pixels Source Image myborder.svg for the Decorative Border with "border-image"

The image with the decorative border can be added using CSS with the following line:

```
...
border: 25px solid transparent;
border-image: url('../images/myborder.svg') 50 50 50 50 round;
...
```

Listing 11.9 /examples/chapter011/11_5_1/css/style2.css

Strictly speaking, `border-image` is again a summarizing property of `border-image-source` (image source), `border-image-slice` (divides the border image into nine parts), `border-image-width` (width of the border), and `border-image-repeat` (determines how the image parts are repeated in the side margins). Figure 11.19 shows the decorative border with `border-image` and the *myborder.svg* graphic from Figure 11.18 in use.



```
.designborder {
 width: 500px;
 padding: 10px;
 border: 25px solid transparent;
 border-image:url('../images/myborder.svg') 50 50 50 50 round;
}
```

Figure 11.19 A Decorative Border with “border-image” (HTML Document Is in /examples/chapter011/11_5_1/index2.html)

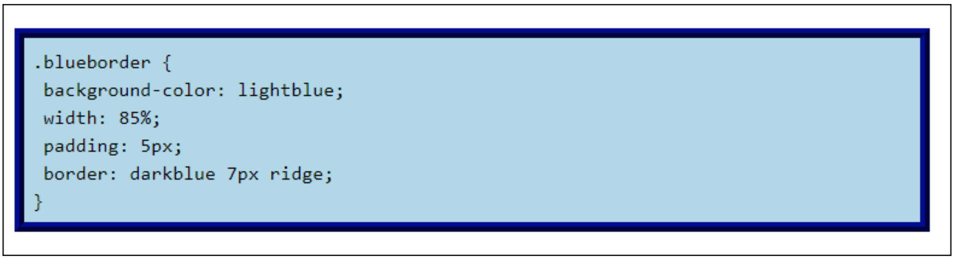
11.5.2 Setting a Background Color Using “background-color”

In the previous examples, you defined the colors for the background as follows:

```
background: lightblue;
```

You can also apply this to boxes. However, I haven’t mentioned yet that `background` is again just a bundling of CSS features for the background, where the first one is equal to `background-color`. Thus, the definition just used leads to the same result as the specification with `background-color`:

```
background-color: lightblue;
```



```
.blueborder {
 background-color: lightblue;
 width: 85%;
 padding: 5px;
 border: darkblue 7px ridge;
}
```

Figure 11.20 The Use of a Background Color within Boxes Can Be Noted Relatively Simply with “background” or “background-color”

The example in Figure 11.20 can be found in */examples/chapter011/11_5_2/index.html* (HTML) and */examples/chapter011/11_5_2/css/style.css* (CSS).

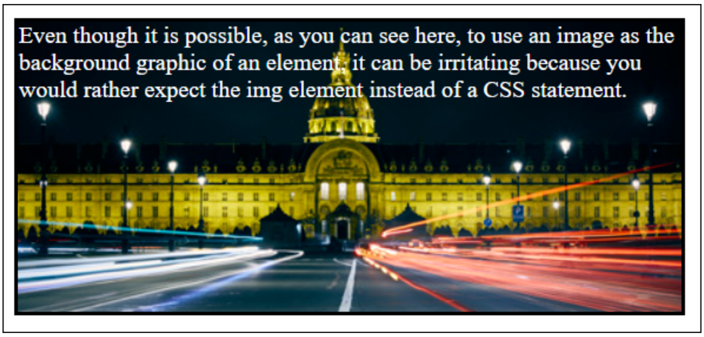
11.5.3 Using Background Images

As you just learned in Section 11.5.2, when you set the background color, `background` is just a bundling or short notation of several CSS features for the background. The following properties are bundled via the short notation, `background`:

- `background-color`
  Background color of the element.
- `background-image`
  Image as background of the element.
- `background-position`
  Position of the background image.
- `background-repeat`
  Repeats the background image along the vertical or horizontal axis.
- `background-attachment`
  Here you can define whether the background can be scrolled or is fixed.

The main focus in this section is on adding a background image inside boxes. It's important to distinguish whether you want to use an image or a background graphic only for decoration. Images or logos should still be added via the HTML element `<img>`. Graphics for design or decoration, on the other hand, can be added as background images using CSS.

Even though it's possible to use an image as the background graphic of an element (here, e.g., as the background image of a `p` element), as you can see in Figure 11.21, it can be confusing because you would expect the HTML element `<img>` rather than a CSS statement. The example can be found in */examples/chapter011/11_5_3/css/style-bg.css*. The HTML document for this is */examples/chapter011/11_5_3/index-bg.html*.



Even though it is possible, as you can see here, to use an image as the background graphic of an element, it can be irritating because you would rather expect the img element instead of a CSS statement.

Figure 11.21 It's Possible, But Rather Untypical, to Use an Image as the Background Graphic of an Element, as Shown Here

You can add a background graphic via `background` in the short notation version or via `background-image`.

Where Do the Background Graphics Come From?

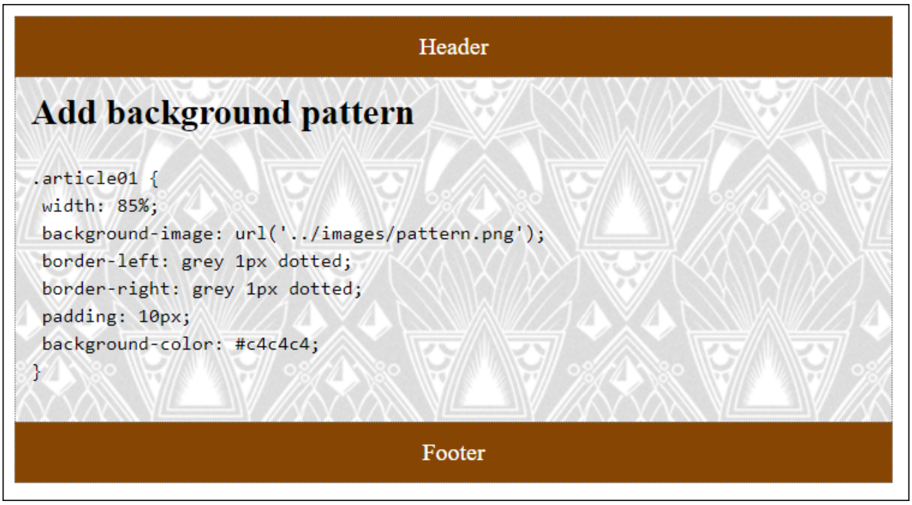
You can either create background graphics yourself or search for ready-made background graphics on the internet. The choice of ready-made background patterns is enormous. For our example, I used a graphic from the website *https://dinpattern.com*.

This is how you insert a background graphic as a class selector for an `article` element:

```
...
.article01 {
   width: 70%;
   background-image: url('../images/pattern.png');
   border-left: gray 1px dotted;
   border-right: gray 1px dotted;
   padding: 10px;
   background-color: #c4c4c4;
}
...
```

Listing 11.10 /examples/chapter011/11_5_3/css/style.css

In the example, the background graphic *pattern.png* is located in the *images* directory of the HTML document. As you can see in Figure 11.22 with */examples/chapter011/11_5_3/index.html* and in the corresponding CSS in */examples/chapter011/11_5_3/css/style.css*, the background graphic also overlays the background color. Nevertheless, you should use a background color that matches the font, as I have done here with `background-color: #c4c4c4;` in case the graphic won't get displayed to a visitor because the user has disabled the graphic display.



Add background pattern

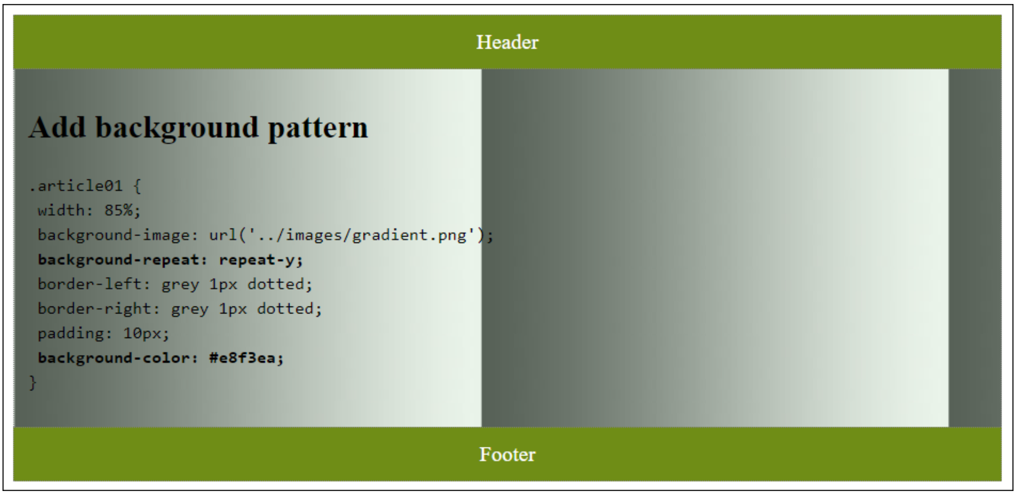
```
.article01 {
  width: 85%;
  background-image: url('../images/pattern.png');
  border-left: grey 1px dotted;
  border-right: grey 1px dotted;
  padding: 10px;
  background-color: #c4c4c4;
}
```

Figure 11.22 A Background Graphic That Overlays the Background Color Has Been Added to the <article> Element (Background Pattern: https://dinpattern.com)

Tiling and Repeating Background Graphics

If you don't specify anything, the background graphic will be repeated (or tiled) vertically and horizontally as many times as there is space available. This is very convenient if you have a suitable background graphic. In Figure 11.23, the pattern was replaced with a 350 × 50 pixel gradient. Toward the bottom, tiling the graphic with a gradient works great, but at the right-hand edge, the gradient doesn't look nice when the background graphic gets repeated.

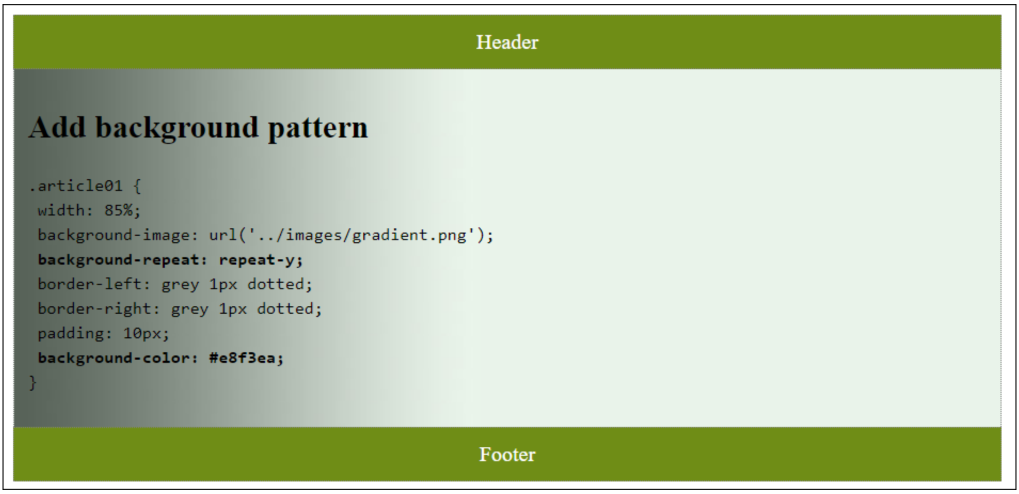
However, you can restrict tiling to the vertical direction and set the background color to be the color at the end of the gradient. For tiling in the vertical direction, you just need to assign the `repeat-y` value to the CSS feature `background-repeat`. You already know how to set the background color. Figure 11.24 shows the result of this modification. The example can be found in */examples/chapter011/11_5_3/index2.html*, while the corresponding CSS file is located in */examples/chapter011/11_5_3/css/style2.css*.



Add background pattern

```
.article01 {
 width: 85%;
 background-image: url('../images/gradient.png');
 background-repeat: repeat-y;
 border-left: grey 1px dotted;
 border-right: grey 1px dotted;
 padding: 10px;
 background-color: #e8f3ea;
}
```

Figure 11.23 Tiling a Background Graphic Doesn't Always Produce the Desired Result



Add background pattern

```
.article01 {
 width: 85%;
 background-image: url('../images/gradient.png');
 background-repeat: repeat-y;
 border-left: grey 1px dotted;
 border-right: grey 1px dotted;
 padding: 10px;
 background-color: #e8f3ea;
}
```

Figure 11.24 Tiling in the Vertical Direction with "background-repeat: repeat-y", and the Matching Background Color Also Works with the Gradient

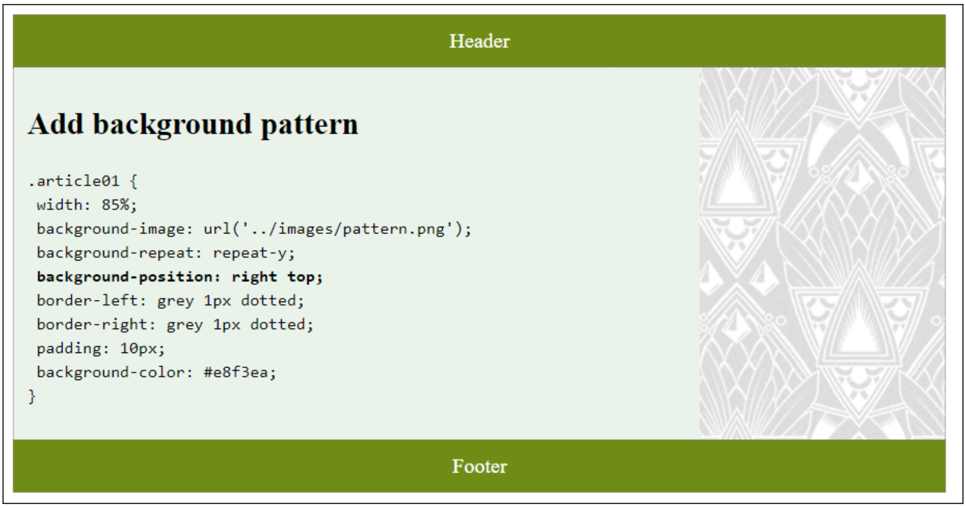
In total, you have three ways to tile a background graphic via the CSS feature `background-repeat`:

- **`background-repeat: repeat-y`**
  Vertical tiling (y-axis).
- **`background-repeat: repeat-x`**
  Horizontal tiling (x-axis).
- **`background-repeat: no-repeat`**
  No tiles at all.

Positioning and Fixing a Background Graphic

You can use the CSS feature `background-position` to position the background graphic in the HTML element. The positions available to you are `top`, `right`, `bottom`, `left`, and `center`. If you use two values, the first value is taken for the horizontal position and the second for the vertical position.

In Figure 11.25, the pattern was positioned with `background-position right top` and tiled with `background-repeat` along the y-axis (`repeat-y`). If you were to use `no-repeat` here, the pattern would only be displayed once in the top-right corner. The example can be found in */examples/chapter011/11_5_3/index3.html*, and the CSS for it is located in */examples/chapter011/11_5_3/css/style3.css*.



Header

Add background pattern

```
.article01 {
 width: 85%;
 background-image: url('../images/pattern.png');
 background-repeat: repeat-y;
 background-position: right top;
 border-left: grey 1px dotted;
 border-right: grey 1px dotted;
 padding: 10px;
 background-color: #e8f3ea;
}
```

Footer

Figure 11.25 The Pattern Was Positioned via “background-position” at the Top Right (“right top”) and Tiled with “background-repeat” along the Y-Axis (“repeat-y”)

The CSS feature `background-attachment` is used pretty rarely; you can use it to fix a background graphic with the value `fixed` to the display area of the page. When scrolling, the graphic remains in place within the HTML element. You can find an example demonstrating this interesting effect when scrolling down in */examples/chapter011/11_5_3/*

index4.html with the corresponding CSS in */examples/chapter011/11_5_3/css/style4.css*. The default value is `scroll`, which scrolls the background graphic along with the element as usual.

The Short Notation for a Background Graphic with "background"

Let's now return to the short notation of `background`: you should have no more trouble bundling the example from */examples/chapter011/11_5_3/css/style4.css* in the short notation, as follows:

```
background: #e8f3ea url('../images/pattern.png') repeat-y right top;
```

This short version thus corresponds to the following:

```
background-image: url('../images/pattern.png');
background-repeat: repeat-y;
background-position: right top;
background-color: #e8f3ea;
```

You've already seen the corresponding result in Figure 11.25.

Stacking Multiple Background Graphics

The stacking of multiple background images can be implemented in CSS as follows:

```
background: url(img01.jpg), url(img02.jpg), url(img03.jpg)
            #e8f3ea;
```

Here, three images are stacked on top of each other. *img01.jpg* is at the top, followed by *img02.jpg* and *img03*.jpg. The background color is set as the last value. The individual background images to be stacked must be separated with commas.

Additionally, you can write the `background-position` and `background-repeat` options. By default, the position is top left, and the tiles are evenly distributed along the y-axis and x-axis. If the images are on top of each other, the upper graphic will cover the one below. For this reason, without a concrete specification of `background-position` and `background-repeat`, only the top background images would be displayed because it would get tiled by default and cover the images below it.

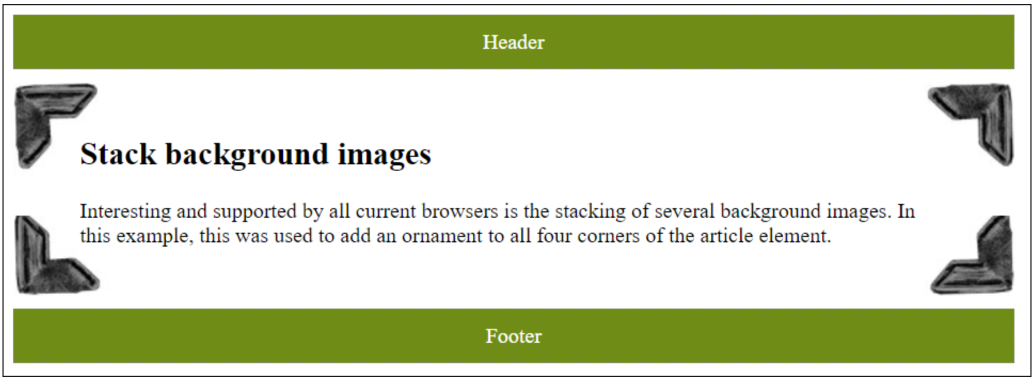
You can implement two background images with different values, for example, as follows:

```
background: url(img01.jpg) left top no-repeat,
            url(img02.jpg) right top repeat-y,
            #e8f3ea;
```

Here, the first background image *img01.jpg* is positioned on the top left and not tiled. The second background image *img02.jpg* is positioned at the top right and tiled down along the y-axis.

In Figure 11.26, with the new ability to stack multiple backgrounds, an ornament was added to all four corners of the `article` element for decoration. The HTML document for this can be found in */examples/chapter011/11_5_3/index5.html*. The CSS statement is as follows:

```
...
background: url('../images/left-top.jpg') top left no-repeat,
            url('../images/right-top.jpg') top right no-repeat,
            url('../images/bottom-right.jpg') bottom right no-repeat,
            url('../images/bottom-left.jpg') bottom left no-repeat,
            white;
...
```



Stack background images

Interesting and supported by all current browsers is the stacking of several background images. In this example, this was used to add an ornament to all four corners of the article element.

Figure 11.26 The Example after Stacking Multiple Background Images and Positioning Them Accordingly without Tiles

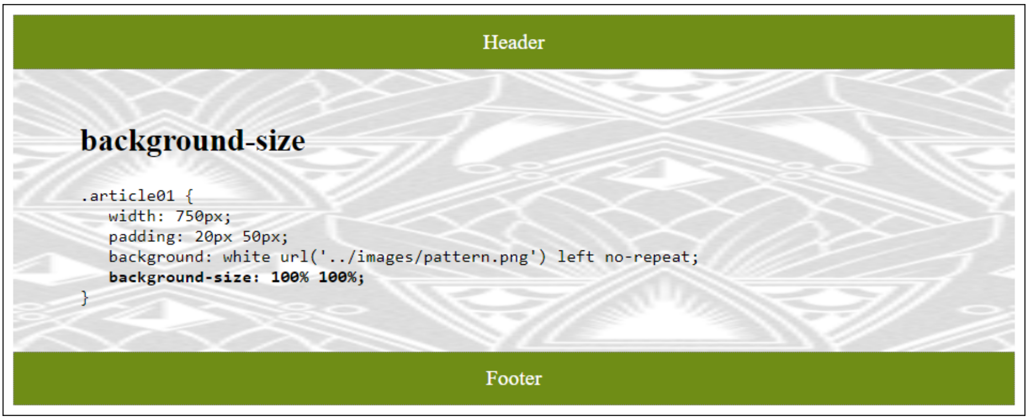
Setting the Size of the Background Image

The CSS feature `background-size` allows you to specify the size of the background image. This enables you to scale the background image up or down, as it were. The information can be given either in pixels or as a percentage. A percentage specification is relative to the height and width of the parent element in which the background image is to be displayed. Let's take a look at the following example:

```
...
.article01 {
    width: 500px;
    padding: 20px 50px;
    background: white url('../images/pattern.png') left no-repeat;
    background-size: 100% 100%;
}
...
```

Listing 11.11 /examples/chapter011/11_5_3/css/style6.css

This makes sure that the background image *pattern.png* completely fills the HTML element that uses the `article01` class. Figure 11.27 shows the */examples/chapter011/11_5_3/index6.html* example with the CSS */examples/chapter011/11_5_3/css/style6.css*; you can see how the background image is stretched across the entire `article` element.



Header

background-size

```
.article01 {
   width: 750px;
   padding: 20px 50px;
   background: white url('../images/pattern.png') left no-repeat;
   background-size: 100% 100%;
}
```

Footer

Figure 11.27 A 189 × 229 pixel Background Image Has Been Stretched Entirely across the <article> Element via “background-size”

It should be clear that upscaling or stretching pixel graphics doesn’t necessarily look nice. There are also two options—`contain` and `cover`—that you can assign to `background-size`:

- **`background-size: contain`**
  This will always display the background image completely inside the box, even if it doesn’t fill the entire surface.
- **`background-size: cover`**
  This will always cover the entire area of the box with the image, even if the image isn’t completely visible.

The Box Model in the Third Dimension

Because the background color and background image have now been added to the box model in addition to the content, `padding`, `border` and `margin`, it’s useful for your understanding to view this box in a third dimension. John Hicks has designed such a 3D box model at *https://hicks.design/journal/3d-css-box-model*.

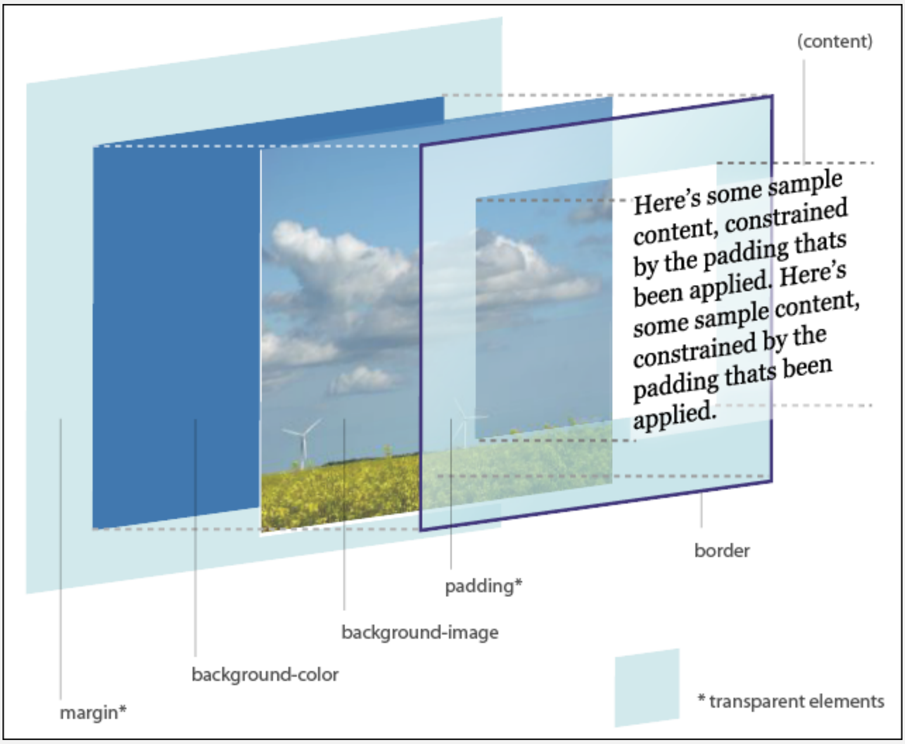

(content)
Here's some sample content, constrained by the padding thats been applied. Here's some sample content, constrained by the padding thats been applied.
border
padding*
background-image
background-color
margin*
* transparent elements

Figure 11.28 The 3D Box Model by John Hicks

11.5.4 Making Boxes Transparent

To create transparent boxes, there are three options available to you. You can use the CSS feature `opacity`, RGBA colors, or HSLA colors, which you already got to know in Chapter 10, Section 10.3.3, with the CSS functions `rgba()` and `hsla()`.

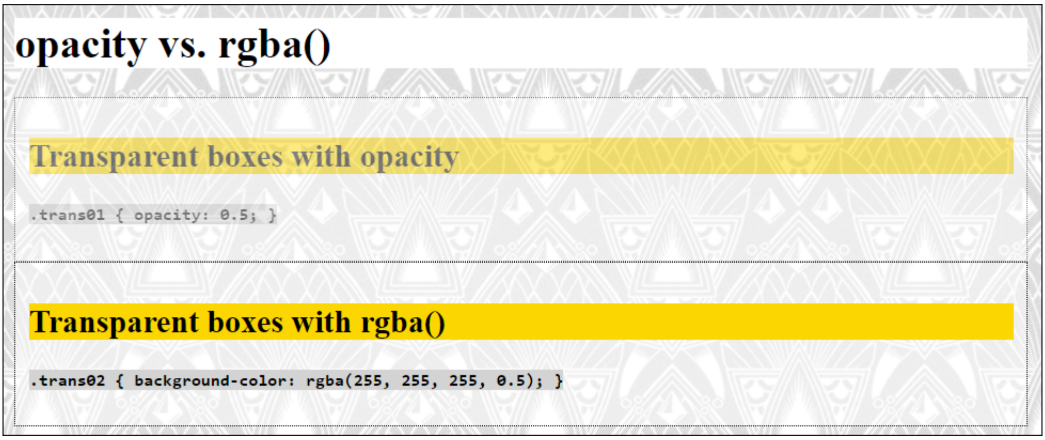
As with RGBA or HSLA colors, the CSS feature `opacity` specifies opacity as `0` for full transparency and `1` for full opacity. The specification `opacity: 0.5;` means that 50% opacity is used.

The difference between `opacity` and the RGBA or HSLA colors is that with `opacity`, the transparency applies to all elements within the box. As you can see in Figure 11.29, where a transparent box with `opacity` has been used in the top `article` element, the font has also become transparent. For the lower `article` element, however, `rgba()` was used, and here only the background becomes transparent. Here's the corresponding code snippet:

```
...
.article_01 {
  width: 90%;
  padding: 10px;
  background-color: white;
  border: black 1px dotted;
  opacity: 0.5;
}
.article_02 {
  width: 90%;
  padding: 10px;
  background-color: white;
  border: black 1px dotted;
  background-color: rgba(255, 255, 255, 0.5);
}
...
```

Listing 11.12 /examples/chapter011/11_5_4/css/style.css

The CSS feature `opacity` is therefore pretty useful for graphics and images because transparency doesn't work with RGBA or HSLA colors. For boxes with text, you should use RGBA or HSLA colors if you want to have a transparent background.



opacity vs. rgba()

Transparent boxes with opacity

```
.trans01 { opacity: 0.5; }
```

Transparent boxes with rgba()

```
.trans02 { background-color: rgba(255, 255, 255, 0.5); }
```

Figure 11.29 The /examples/chapter011/11_5_4/index.html Example with Transparent Boxes: One with "opacity" and One with "rgba()"

11.5.5 Adding a Gradient

CSS also allows you to create and use a gradient without having to insert a background image. You must create a linear gradient using the CSS function `linear-gradient()` and assign it to the `background` (or `background-image`) property. The easiest way to do this is as follows:

```
background: linear-gradient(white, orange);
```

This displays a linear gradient from white to orange from top to bottom (default setting).

To direct the gradient in a different direction, you just need to specify the keyword `to`, followed by the direction `bottom` (top to bottom), `top` (bottom to top), `right` (left to right), or `left` (right to left), for example:

```
background: linear-gradient(to right, white, orange);
```

This displays a linear gradient from white to orange, but this time, a gradient from left to right has been defined via `to right`. You can define diagonal directions via `to right bottom` (top left to bottom right) and `to left top` (bottom right to top left). It's also possible to specify the position of the color break. A line without specifying a color break appears as follows:

```
background: linear-gradient(to right, white, orange);
```

The specification for the color break with values for the gradient looks as follows:

```
background: linear-gradient(to right, white 0%, orange 100%);
```

A line with the specification of a color break appears as follows:

```
background: linear-gradient(to right, white 30%, orange 70%);
```

You would therefore display a color gradient from left to right. The first color break of white is performed after 30%. The second color break of orange is performed after 70%. At each of these color breaks, the corresponding color starts and continues until the next color break (or until the end, if there's no further color break). You would therefore create a hard transition if both color breaks were in the same place, as shown in the following example:

```
background: linear-gradient(to right, white 50%, orange 50%);
```

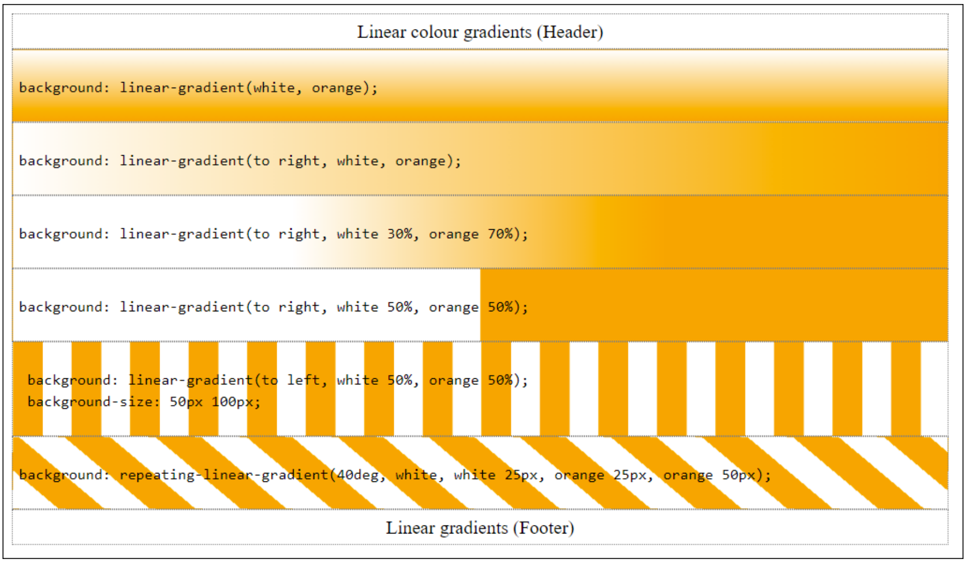
Here, the color transition is shown exactly after 50% with a hard edge because both color breaks were specified there. If you combine this information with `background-size`, you can create a pattern:

```
background: linear-gradient(to left, white 50%, orange 50%);
background-size: 50px 100px;
```

Figure 11.30 shows all the examples described here in use. Instead of specifying color names, you can also use hex values, RGB/HSL colors, or RGBA/HSLA colors.

In Figure 11.30, we also demonstrated the CSS function `repeating-linear-gradient()`, which allows you to repeat a gradient. The example snippet follows:

```
background: repeating-linear-gradient(40deg,
            white, white 25px, orange 25px, orange 50px);
```



Linear colour gradients (Header)

```
background: linear-gradient(white, orange);
background: linear-gradient(to right, white, orange);
background: linear-gradient(to right, white 30%, orange 70%);
background: linear-gradient(to right, white 50%, orange 50%);
background: linear-gradient(to left, white 50%, orange 50%);
background-size: 50px 100px;
background: repeating-linear-gradient(40deg, white, white 25px, orange 25px, orange 50px);
```

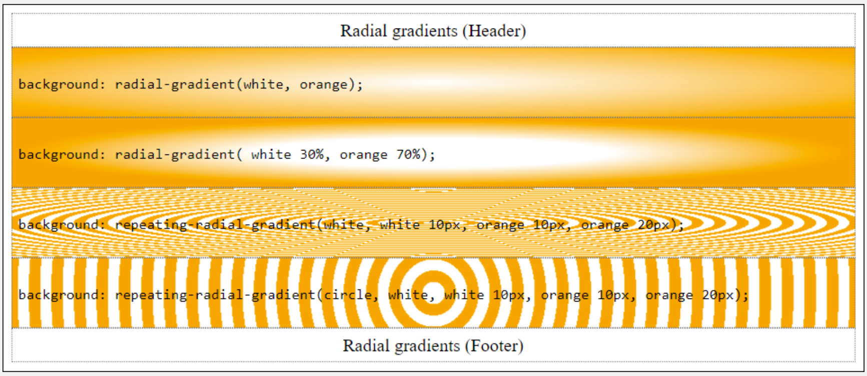
Linear gradients (Footer)

Figure 11.30 Linear Gradients with "linear-gradient()" (Example in /examples/chapter011/11_5_5/index.html, and CSS File in /examples/chapter011/11_5_5/css/style.css)

In this example, you use a gradient at a 40-degree angle (`40deg`), starting with white color and having a width of 25 pixels, while the color orange also starts exactly at the position (`25px`), ending after 50 pixels. Then the gradient gets repeated.

Radial Gradients

In addition to the linear CSS gradients `linear-gradient()` and `repeating-linear-gradient()` mentioned here, there are two counterparts that enable you to create radial gradients from the center point. The radial gradients can be created via the CSS functions, `radial-gradient()` or `repeating-radial-gradient()`. You can see examples of this in Figure 11.31. The corresponding example files can be found in */examples/chapter011/11_5_5/index2.html* and */examples/chapter011/11_5_5/css/style2.css*.



Radial gradients (Header)

```
background: radial-gradient(white, orange);
background: radial-gradient( white 30%, orange 70%);
background: repeating-radial-gradient(white, white 10px, orange 10px, orange 20px);
background: repeating-radial-gradient(circle, white, white 10px, orange 10px, orange 20px);
```

Radial gradients (Footer)

Figure 11.31 The Example with Radial Color Gradients in Use

You can use several colors for the gradient. A linear or radial gradient in a rainbow color can be created as follows:

```
...
background: linear-gradient(to left,
   purple, blue, green, yellow, red, purple);
background: radial-gradient(
   purple, blue, green, yellow, red, purple);
...
```

Listing 11.13 /examples/chapter011/11_5_5/css/style3.css

Creating Gradients with Online Tools

Those who find the manual creation of gradients with CSS a bit too cumbersome should take a look at the online tool *Ultimate CSS Gradient Generator* at *www.colorzilla.com/gradient-editor/*. This online tool enables you to generate a color gradient like you would with Photoshop. The tool generates the CSS code from this, which you copy and paste into your project.

11.5.6 Adding a Shadow Using the “box-shadow” Feature

In the past, if you wanted to add shadows to the elements, you had to do it by means of graphics. For websites with a flexible width, that was hardly useful. The CSS feature `box-shadow` makes adding shadows a breeze. The simplest specification to create a shadow with the CSS feature `box-shadow` is as follows:

```
box-shadow: 4px 4px gray;
```

This allows you to place a gray shadow around an element with a horizontal and vertical offset of 4 pixels each. The first value is used for the horizontal shift (*offset-x*) and the second value for the vertical one (*offset-y*). A positive value shifts the shadow of the horizontal shift to the right and the vertical shift down. The opposite happens with negative values. Instead of a color name, you can also use a hexadecimal color specification or the RGB(A) or HSL(A) color specifications.

If you want to have a soft shadow, you only need to specify a third value that defines the *blur* of the shadow. The higher the value, the softer the shadow will be (default value is `0`):

```
box-shadow: 4px 4px 4px gray;
```

If you still want to determine the radius or the spread of the shadow, you can specify that with a fourth value:

```
box-shadow: 4px 4px 4px 4px gray;
```

The last value defines the color of the shadow, but it can also be in front of the other values.

In Figure 11.32, you'll see the shadows described here in actual use. In addition, you can also see an example with an inner shadow that you can define and use via `inset` as the first value of `box-shadow`. The CSS code for this can be found in */examples/chapter011/11_5_6/css/style.css*.

Shadow Generator

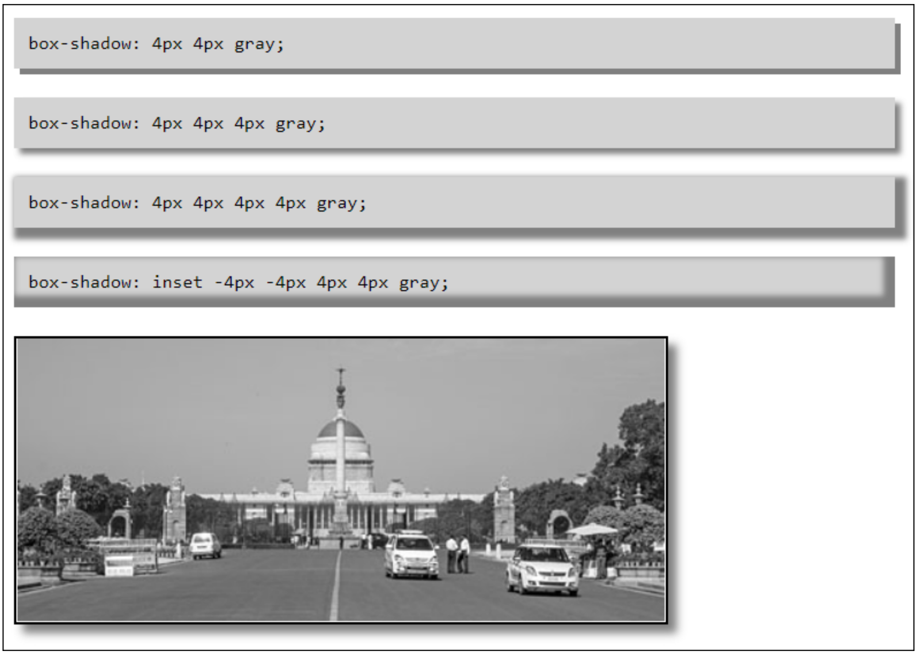
If you don't feel like creating the shadow manually, you can find various generators on the web that do this work for you, such as *html-css-js.com/css/generator/box-shadow/*.

It's also possible to specify multiple shadows at once, separated by commas, for one element, for example:

```
box-shadow: 4px 4px 4px red, -4px 4px 4px yellow,
            4px -4px 4px green, -4px -4px 4px blue;
```

In this example, four shadows are defined at once.

As you may recognize in Figure 11.32, a shadow doesn't extend the area around the elements in the way `margin` does. If the shadow is extremely large, it can extend into the next element.



```
box-shadow: 4px 4px gray;
box-shadow: 4px 4px 4px gray;
box-shadow: 4px 4px 4px 4px gray;
box-shadow: inset -4px -4px 4px 4px gray;
```

Figure 11.32 Adding Shadows for HTML Elements Becomes a Breeze Thanks to "box-shadow" (Example in /examples/chapter011/11_5_6/index.html)

11.5.7 Adding Round Corners Using the CSS Feature "border-radius"

For round corners of elements, you can use `border-radius`. This CSS feature can be used independently of `border`. You don't have to draw an extra frame to round off corners. In turn, the use of `border-radius` is a short notation of `border-top-left-radius`, `border-top-right-radius`, `border-bottom-right-radius`, and `border-bottom-left-radius`. For example, you can write a `border-radius` as follows:

```
border-radius: 20px;
```

This is a shortened notation of the following:

```
border-top-left-radius: 20px;
border-top-right-radius: 20px;
border-bottom-right-radius: 20px;
border-bottom-left-radius: 20px;
```

You can also specify two, three, or four values for `border-radius`. Here's an example with two values:

```
border-radius: 20px 10px;
```

The upper-left and lower-right corners get a radius of 20 pixels, and the upper-right and lower-left corners get a radius of 10 pixels. The order for using all four values is top left, top right, bottom right, and bottom left, which is similar to the short notation of `padding` or `margin`, that is, clockwise, starting at the top left.

In Figure 11.33, I've rounded off the top corners of the header with the following information:

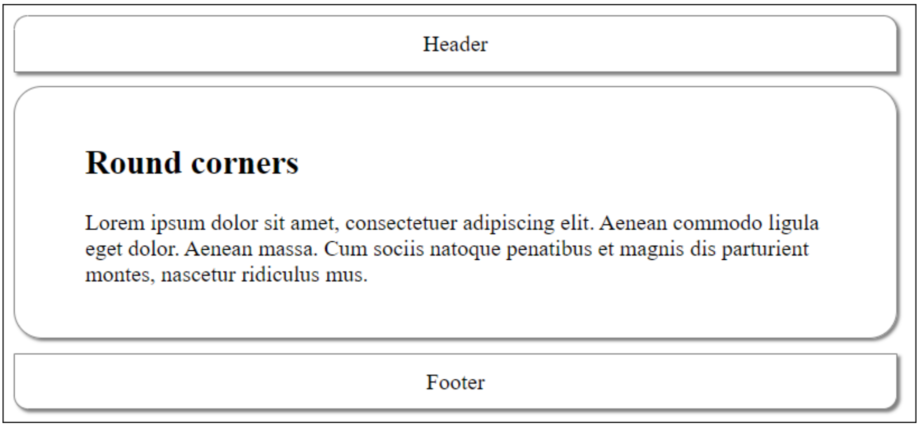
```
border-top-left-radius: 10px;
border-top-right-radius: 10px;
```

The `article` element in the middle, on the other hand, has been rounded off with a simple `border-radius` of 20 pixels for all sides. In the last example with the footer, I rounded off the bottom corners with the following short notation:

```
border-radius: 0px 0px 10px 10px;
```

Instead of pixels (`px`), you can also use other units such as percent (`%`), `em`, or `rem`.

`border-radius` can be applied nicely to images, as you can see in Figure 11.34. The image on the left has a `border-radius` of 25 pixels. The image on the right was shaped into an ellipse by setting the `border-radius` to 50%. The example can be found in */examples/chapter011/11_5_7/index2.html*; the CSS code is contained in */examples/chapter011/11_5_7/css/style2.css*.



Header

Round corners

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Footer

Figure 11.33 Round Corners Are Relatively Easy to Create (Example in /examples/chapter011/11_5_7/index.html; CSS Is in /examples/chapter011/11_5_7/css/style.css)

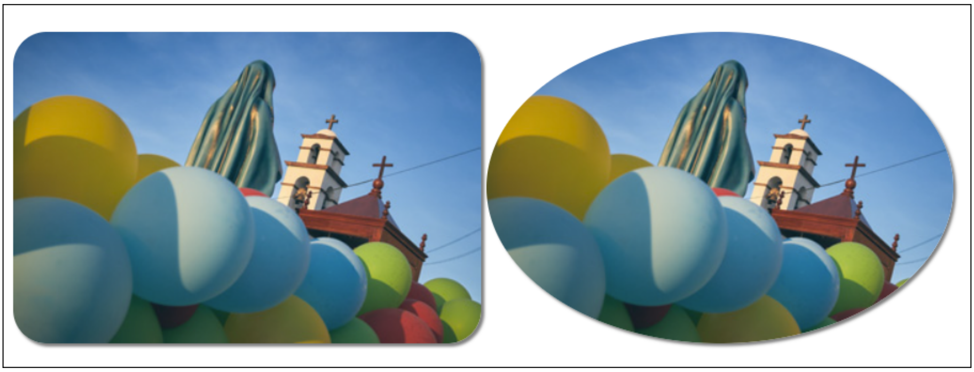


Figure 11.34 “border-radius” Applied to Images

You also have the option to specify different values for the horizontal and vertical radius. To do that, you need to separate these two values with a slash. As a result, you get corners with elliptical curves, for example:

```
border-radius: 5px / 20px;
```

This sets an ellipse of 5 pixels × 20 pixels for all corners. Of course, you can use the same as a percentage:

```
border-radius: 80% / 20%
```

If you want to set each corner individually, you can use `border-top-left-radius`, `border-top-right-radius`, `border-bottom-right-radius`, and `border-bottom-left-radius` as before. The short notation can also be used here as follows:

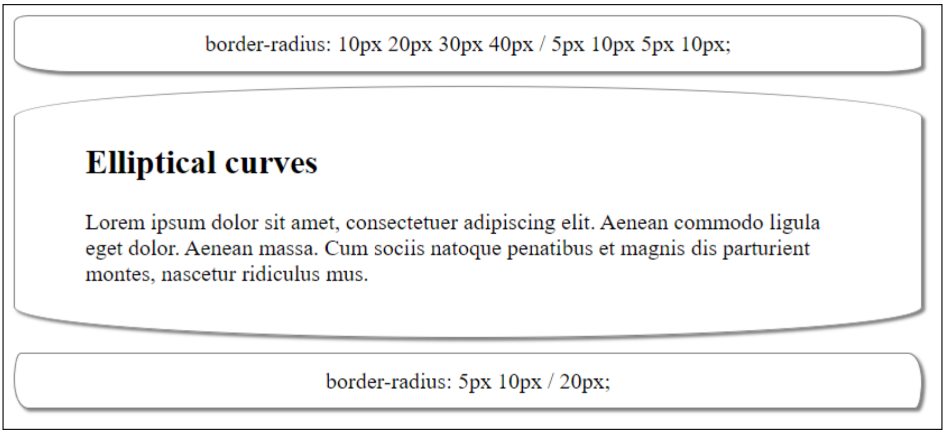
```
border-radius: 5px 10px / 20px;
```

This notation corresponds to the following:

```
border-top-left-radius: 5px / 20px;
border-top-right-radius: 10px / 20px;
border-bottom-right-radius: 5px / 20px;
border-bottom-left-radius: 10px / 20px;
```

You can also use the short notation for all corners differently as follows:

```
border-radius: 10px 20px 30px 40px / 5px 10px 5px 10px;
```



border-radius: 10px 20px 30px 40px / 5px 10px 5px 10px;

Elliptical curves

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

border-radius: 5px 10px / 20px;

Figure 11.35 With "border-radius", You Can Also Provide Elements with Elliptical Curves

Thus, you've achieved the same result as with the following notation:

```
border-top-left-radius: 10px / 5px;
border-top-right-radius: 20px / 10px;
border-bottom-right-radius: 30px / 5px;
border-bottom-left-radius: 40px / 10px;
```

You can see the example for this in Figure 11.35 and find it in */examples/chapter011/11_5_7/index3.html*; the CSS code is located in */examples/chapter011/11_5_7/css/style3.css*.

11.6 Related Topic: Web Browser Prefixes (CSS Vendor Prefixes)

CSS, like HTML, is constantly evolving, and web browser vendors are doing their best to release newer versions with new features at ever-shorter intervals. As a web developer, this rapid progress also benefits you because many web browser vendors like to implement new experimental CSS features in the very early state of the standard version.

With the help of a special web browser prefix or *vendor prefix*, you can thus already use CSS features *experimentally* that are still in draft or beta state. Once the corresponding

CSS module is in the final version, and the web browser fully supports the feature, you can remove the web browser prefix. Alternatively, you can leave it in place, allowing older web browsers to use this feature, if necessary, if the CSS feature is already included in the final *recommendation*.

From Working Draft to Recommendation

The development process toward a finished W3C Recommendation is divided into several stages. It all usually starts with a *working draft*. Often, multiple working drafts are developed and not all make it to the recommendation stage. The next stage is the *last call working draft*, which is something like the last planned working draft. Once the working drafts are done, the next stage is the candidate recommendation, where technical details of the new technology are already known. The next-to-last stage is the *proposed recommendation*, where additional implementations can be added to the existing ones. This stage is also the last stage at which one can still influence the development process. If the members finally agree to the recommendation proposal, the proposal is given the final status of a *recommendation*. Recommendations may be withdrawn for revision at any time.

As a simple example, we'll use the CSS feature `text-emphasis`, which is to be newly introduced in CSS selectors level 4 and can only be used experimentally in some web browsers (at the time of print) using web browser prefixes. This CSS feature allows you to highlight the emphasis of text with a different color and style, for example, with circles, points, or triangles. However, this isn't about the feature in detail, but about the browser prefixes. Here's an example:

```
h1 {
    text-emphasis: filled double-circle blue;
}
```

This is used to particularly highlight the `h1` heading with blue double circles if the web browser can handle the CSS feature `text-emphasis`, which wasn't the case when this book went into print. However, because some web browsers have already implemented this CSS feature experimentally, you can use and test browser prefixes as follows:

```
...
h1 {
    -moz-text-emphasis: filled double-circle blue;
    -webkit-text-emphasis: filled double-circle blue;
    text-emphasis: filled double-circle blue;
}
```

Listing 11.14 /examples/chapter011/11_6/css/style.css

The procedure is relatively simple. You introduce such experimental CSS features with a vendor-specific abbreviation (e.g., with `-moz-` for Mozilla Firefox). Then you can test these CSS features and feed your experience with them back to the web browser manufacturers, if necessary. This vendor-specific CSS feature is understood only by a particular web browser or web browser family.

If a web browser can't do anything with the web browser prefix features, you don't need to bother about it because this feature gets ignored anyway. The web browser itself picks out what it knows and can do. And if a web browser doesn't know `text-emphasis` at all, nothing of the sort will be used, and you may need to provide an alternative for such a web browser.

See Table 11.3 for a list of common web browser prefixes.

Code	Vendor
`-webkit-`	Chrome, Safari, Edge
`-moz-`	Mozilla Firefox
`-o-`	Opera

Table 11.3 List of Common Web Browsers and Their Prefixes

Regarding such examples, it should be noted that you're nevertheless responsible for compliance with the standard version yourself. This means that a browser-specific CSS feature that you've written with a web browser prefix can change again or perhaps be deleted altogether.

Using or Not Using Web Browser Prefixes?

The web browser prefixes make the CSS code ugly and more voluminous. Nevertheless, they've become a favorite tool of web developers. It often happens that you don't even know whether a certain CSS module has already been fully implemented or not. It isn't always easy to keep track of everything, and there's a risk that the web browser hasn't even fully implemented the feature yet. However, you have the advantage of being able to use a feature of the future standard version that isn't yet complete. A good overview of this is provided at *http://caniuse.com*. There you'll find tables about the current versions and about how far certain web browsers support a certain CSS feature (and also HTML feature) and/or from which version the web browser prefix can or should be used.

11.7 Summary

This chapter was completely dedicated to rectangular boxes. You've learned about the following:

- The classic box model of CSS
- The newer alternate `border-box` box model of CSS
- How to design these boxes with features such as background color, background graphic, transparency, gradient, shadow, and round corners

Chapter 12

CSS Positioning

You're now familiar with the box model of CSS. However, before you can start creating layouts with CSS, you're still missing one small important building block. The question is how you can position HTML elements or the boxes with CSS.

Until now, you've been used to HTML elements being positioned and displayed in the flow of the HTML document. The elements are displayed one after the other in the order in which they were noted in the HTML code. However, you aren't limited to such static positioning and can manipulate this using CSS.

There are several ways to perform positioning tasks with CSS. Here's what you'll learn in this chapter:

- **Positioning model**
  Positioning HTML elements using the CSS feature `position`.
- **Stacking model**
  Stacking HTML elements using `z-index`.
- **Float model**
  Floating HTML elements with the CSS feature `float`.
- **Flexbox model**
  Positioning HTML elements with flexible boxes.

12.1 Positioning via CSS Feature "position"

The CSS feature `position` allows you to determine how and where you position an element and what should happen to the elements following it. In this context, a distinction is made between four possible methods for positioning elements:

- **Static positioning (`position: static;`)**
  This is the default setting for all elements, and it gets used if you haven't written the CSS feature `position` at all. In the process, all elements are arranged one after the other as usual, as they were written in the HTML document.
- **Relative positioning (`position: relative;`)**
  This method places or moves an element relative to its current position with the CSS features `top`, `bottom`, `left`, and `right` and the corresponding value specifications. The

other elements won't get affected by this; that is, the other elements remain in the same position as if the moved element would remain in its original place.

- **Absolute positioning (`position: absolute;`)**
  You can use this method to drag the element out of the document flow. Here, you can use the CSS features `top`, `bottom`, `left`, and `right` to place the element absolutely in the nearest parent element or web browser window, regardless of where the HTML element was written in the HTML document. All other elements then act as if the absolutely moved element no longer belongs to the document flow, and any gap thus created is "filled" with the element that follows next, or it moves up.
- **Fixed positioning (`position: fixed;`)**
  Initially, the fixed positioning behaves like the absolute positioning, but with the clear difference that this fixed position is measured absolutely to the upper-left edge of the web browser window. In practice, unlike the absolute position, this means that a fixed element no longer moves when the web browser window gets scrolled.
- **Sticky positioning (`position: sticky;`)**
  This function is a hybrid of relative and fixed positioning. The element will initially behave similar to relative positioning, until a certain boundary such as the top or bottom of the screen is reached, where the element will then "stick" and behave similar to fixed positioning.

12.1.1 Normal Positioning ("position: static")

Unless you've specified otherwise, the elements are arranged according to the corresponding box type (block or inline) in the document flow. Basically, you've always used the positioning with `position: static;` in the book so far because this is the default setting. When you use this method, the elements get displayed one after the other in the document flow, that is, as they were written in the HTML document.

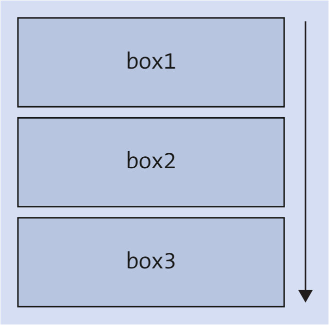


Figure 12.1 Static Positioning with "position: static" as the Default Setting. Each New Element Follows the Other as It Was Written in the HTML Document.

The document flow (often referred to as *flow*) for block elements (or block boxes) usually starts as far to the top left as possible and is as wide as the surrounding box. The next box starts in the next line. The principle behind this scenario is shown in Figure 12.1. Inline elements (or inline boxes), on the other hand, also start at the top left, but become only as wide as the content. The next inline element gets positioned to the right of it. If there isn’t enough space left on the right-hand side, the element slides to the next line. The question as to why the rather inappropriate keyword `static` was used here instead of a more appropriate one like *flow* remains a mystery.

Here’s a simple code snippet for this:

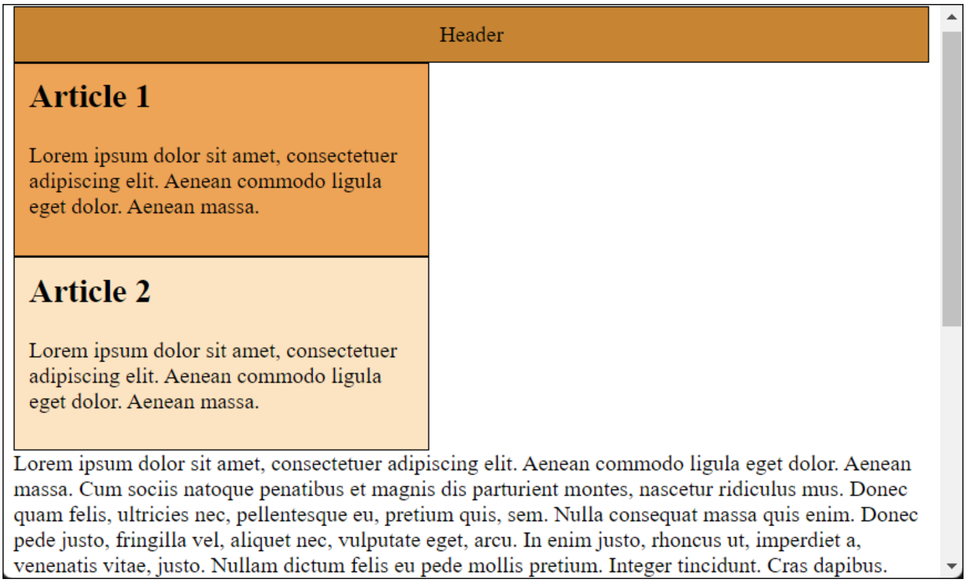
```
...
.article01 {
  position: static;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: sandybrown;
}
.article02 {
  position: static;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: bisque;
}
...
```

Listing 12.1 /examples/chapter012/12_1/css/style.css

```
...
  <body>
    <header class="foothead">Header</header>
      <article class="article01">
        <h1>Article 1</h1>
        <p>Lorem ipsum dolor sit amet ...</p>
      </article>
      <article class="article02">
        <h1>Article 2</h1>
        <p>Lorem ipsum dolor sit amet ... </p>
      </article>
       ...
    <footer class="foothead">footer</footer>
  </body>
...
```

Listing 12.2 /examples/chapter012/12_1_1/index.html

In this example, you could have omitted the lines `position: static;` because this is the default setting anyway. For this reason, there's nothing surprising in Figure 12.2: The individual elements are displayed one after the other, as written in the document flow.



Header

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus.

Figure 12.2 Default Static Arrangement according to the Document Flow with the Default Setting “position: static;”

12.1.2 Positioning Elements Using “top”, “right”, “bottom”, and “left”

Before we go into the details of relative, absolute, and fixed positioning, I want to briefly describe the CSS features, `top`, `right`, `bottom`, and `left` (also referred to as *offset properties*). As a matter of fact, these four CSS features are also used for the CSS feature `position`. This way, you can specify whether an absolutely or relatively positioned element starts at the `top`, `right`, `bottom`, and/or `left`. For a normal positioning with `position: static;`, specifications with the CSS features `top`, `right`, `bottom`, and `left` have no effect. In that case, the default orientation is always as far up on the left as possible. Common units for the `top`, `right`, `bottom`, or `left` values are pixel (`px`), percent (`%`), and `em`.

It makes sense to use one property each of the values `top` and `bottom` or `left` and `right`. Specifications of top and bottom or left and right are rarely useful in practice, and a second specification is ignored if it doesn't correspond to the (also allowed) negative value of the first value.

The default value of all four CSS features is `auto`, which means that the position of the corresponding edges depends on the surrounding elements, or more simply, contains no special starting position.

12.1.3 Relative Positioning (“position: relative”)

Relative positioning moves an element from its current position. No other elements are affected by this move and they remain in the same place, as if the moved element were also still in the same place. Because the other elements aren’t affected, relatively placed elements can overlay other elements.

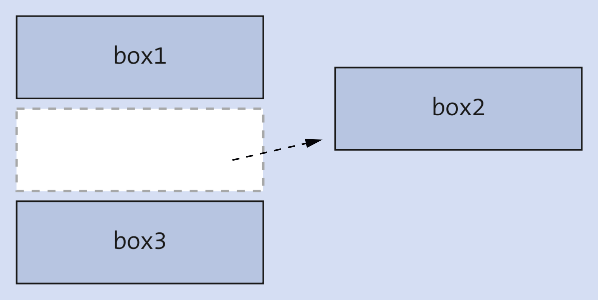


Figure 12.3 Relative Positioning Moves the Element Relative from the Static Position: Subsequent Elements Behave as If the Element Hadn’t Been Positioned

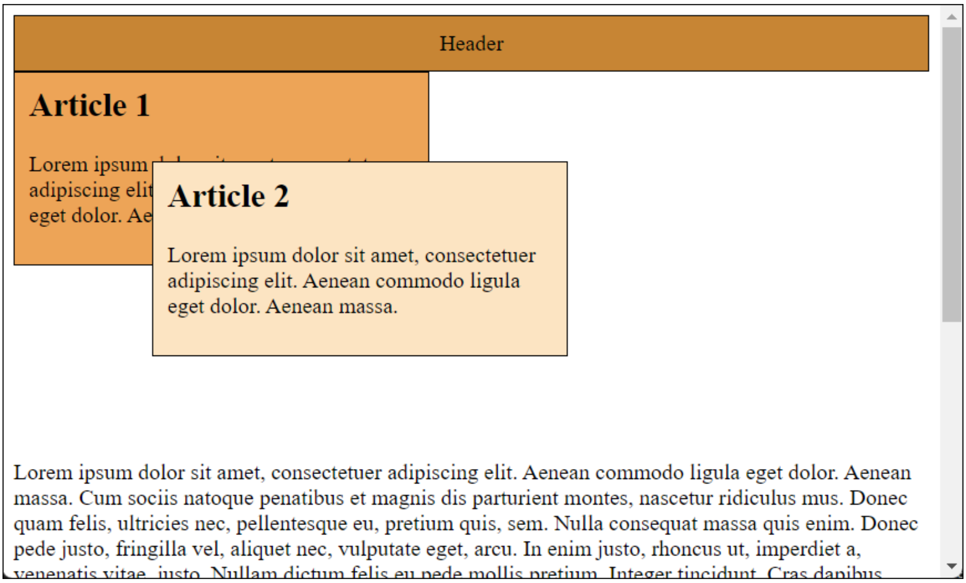
For this purpose, the example from */examples/chapter012/12_1/css/style.css* is modified slightly to demonstrate relative positioning:

```
...
.article01 {
  position: static;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: sandybrown;
}
.article02 {
  position: relative;
  top: -75px;
  left: 100px;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: bisque;
}
...
```

Listing 12.3 /examples/chapter012/12_1_3/css/style.css

Here, in the second class selector `article02`, you’ve set the position to `relative` using the CSS feature `position`. This specification now causes the element using this class with `class=article02` to move from the current position of `top` relatively 75 pixels up and from `left` 100 pixels to the right (see Figure 12.4).

In Figure 12.4, you can see how the paragraph text and all other elements behind the `Article 2` element do *not* get changed in their position. All other elements behave as if the moved element was still in its old place. The example can be found in */examples/chapter012/12_1_3/index.html*.



Header

Article 1

Lorem ipsum
adipiscing elit
eget dolor. Ae

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus.

Figure 12.4 Relative Positioning with CSS Offsets the Element with “top”, “bottom”, “right”, and “left” Relative to Itself, and the Gap in the Document Flow Remains

Positive values move the element inward from the corresponding specified edge, while negative values move it outward. You could have achieved the same position via the CSS features, `bottom` and `right`:

```
...
bottom: 20px;
right: -50px;
...
```

12.1.4 Absolute Positioning (“position: absolute”)

When using absolute positioning, the element gets pulled out of the ordinary document flow. In contrast to relative positioning, all other elements react as if the absolutely shifted element didn’t exist at all. As a result, there’s no longer a gap between the moved and the following elements, as was previously the case with `position: relative;`. Here, too, the position gets defined with `top`, `bottom`, `left`, and `right`.

The absolute position here is the upper-left corner of the parent element, which can be positioned using the absolute, fixed, or relative method. If no parent element exists, the position information refers to the top element in the document tree, which is `<html>`.

In Figure 12.5, you can see that with absolute positioning the element has been moved relative to the enclosing parent element. If there’s no parent element, the element is aligned relative to the *viewport*, that is, the web browser window.

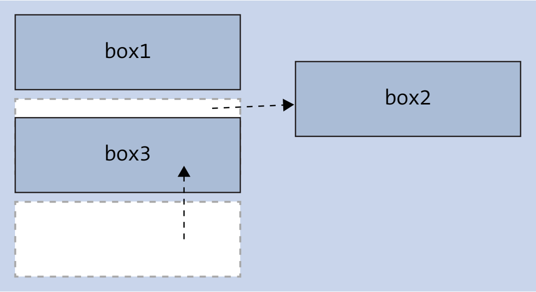


Figure 12.5 Absolute Positioning Moves the Element Relative to the Enclosing Parent Element

In this case, the element is removed from the usual document flow, so that the subsequent elements move up as if there had never been anything in that position.

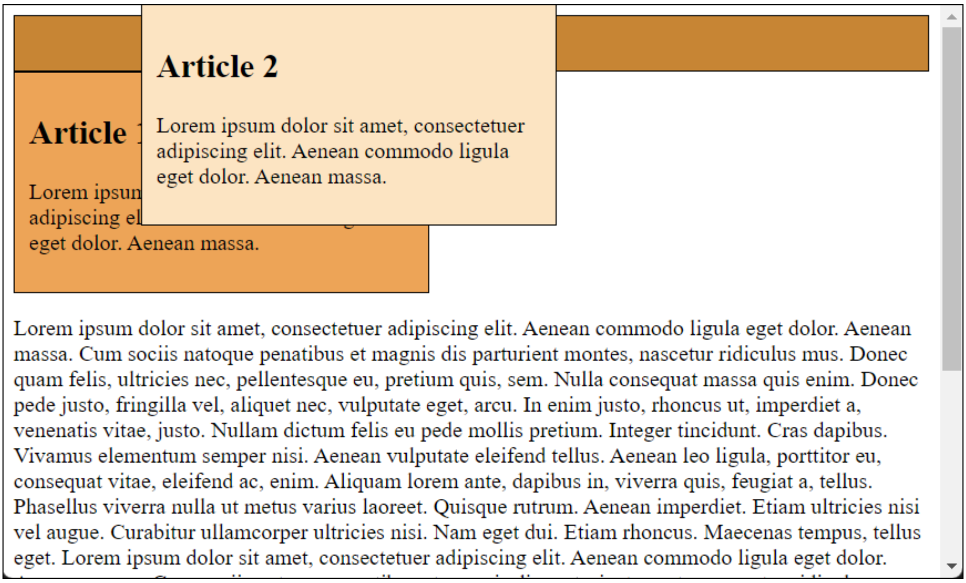
Take a look at the example with two articles, where the second `article` element has been positioned via the absolute method:

```
...
.article01 {
  position: static;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: sandybrown;
}
.article02 {
  position: absolute;
  top: 0;
  left: 100px;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: bisque;
}
...
```

Listing 12.4 /examples/chapter012/12_1_4/css/style.css

By specifying `absolute` in the `article02` class selector, the element that uses this class with `class=article02` gets completely lifted out of the document flow and is effectively no longer available to the other elements. In the example, the element was moved 0 pixels from the top, that is, not at all, and 100 pixels from the left. The result in Figure 12.6 shows the effect of 0 pixels from the top of the web browser window (or more precisely, the `html` element) and 100 pixels from the left-hand side of the web browser window.

In addition, you can see that the box is virtually free-floating over the other elements of the web page. The other elements also ignore this box entirely and fill the gap with the element following it in the document flow. In the example, the gap was filled with the following paragraph text by moving it up. The HTML document to run can be found in */examples/chapter012/12_1_4/index.html*.



Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Article

Lorem ipsun
adipiscing el
eget dolor. Aenean massa.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

Figure 12.6 The Absolutely Positioned Area Floats Completely Detached above the Web Page

Attention with "position: absolute;" without Width Specification!

If you don't specify the width of a block and use `position: absolute;` for positioning, this absolutely positioned element will get displayed with the same width as the content.

In practice, absolute and relative positioning are often combined. A commonly used scenario here is to place a caption over an image, which can be easily done with `<figcaption>` as the child element and `<figure>` as the parent element.

```
...
  <figure>
    <img src="./images/surfer.jpg" alt="Surfer on Snake River" width="400">
    <figcaption>Surfer on Snake River</figcaption>
  </figure>
...
```

Listing 12.5 /examples/chapter012/12_1_4/index2.html

First, you have to position the parent element `<figure>` relative to `position: relative;`. This positioning has no effect yet. `<figcaption>`, on the other hand, can be lifted out of the document flow using `position: absolute;`. Once you do that, you can position the caption within `<figure>` using `left`, `right`, `bottom`, or `top`—depending on where you want the caption to go. I’ve set `left` and `top` to `0` here, which places the caption at the top. If you want to place it at the bottom, you just need to use `bottom: 0` instead of `top: 0`. Here’s the CSS for that (see Listing 12.6).

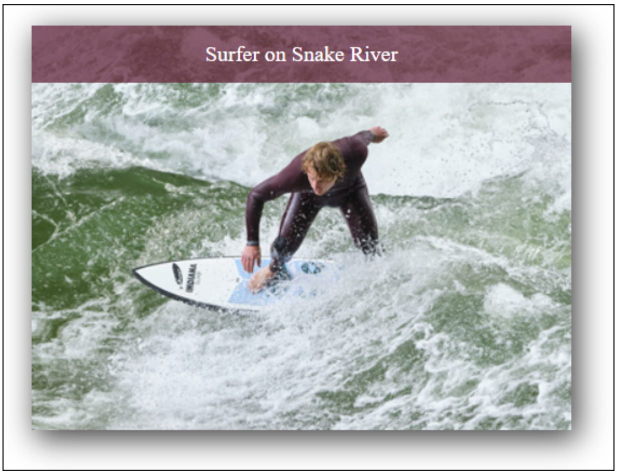


Figure 12.7 Using the Combination of an Absolute and a Relative Position, the Image Caption was Added Here Easily and Quickly

```
...
figure {
    position: relative;
    width: 400px;
    box-shadow: 4px 6px 22px 1px rgba(0, 0, 0, 0.75);
}

figcaption {
    position: absolute;
    left: 0;
    top: 0;
    width: 100%
    text-align: center;
    color: white;
    background: rgba(100, 28, 52, 0.7);
    padding: 0.75rem;
}
...
```

Listing 12.6 /examples/chapter012/12_1_4/css/style2.css

12.1.5 Fixed Positioning ("position: fixed")

Fixed positioning corresponds to the principle of absolute positioning with the `absolute` value described in the previous section, the only difference being that here an element is fixed and thus no longer scrolled. This way, you practically align the element rigidly to the web browser window (*viewport*).

Again, you can see in Figure 12.8 how, in the case of fixed positioning, the element is pulled out of the document flow and positioned absolutely, that is, moved relative to the comprehensive element, and the elements following it in the document flow move up. Basically, everything reminds us of absolute positioning; the only difference is that this repositioned element remains fixed and can no longer be scrolled.

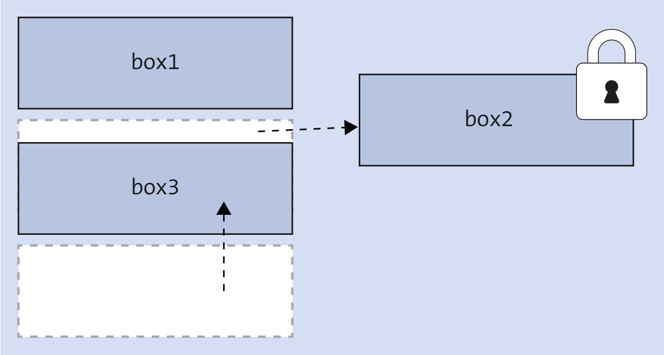


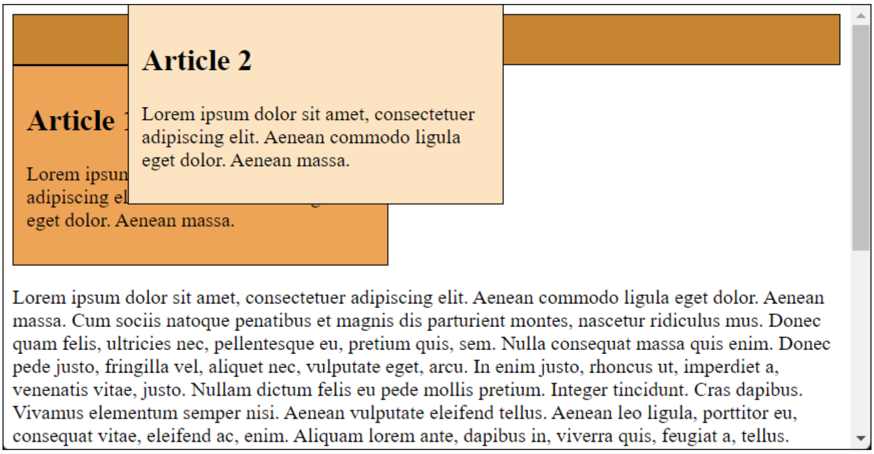
Figure 12.8 For Fixed Positioning, the Element Gets Pulled Out of the Document Flow and Positioned Absolutely. The Only Difference Is That This Element Remains Fixed.

For this purpose, all you need to do is set the CSS feature `position` from `absolute` to `fixed` in */examples/chapter012/12_1_4/css/style.css* from the previous section:

```
...
.article02 {
  position: fixed;
  top: 0;
  left: 100px;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: bisque;
}
...
```

Listing 12.7 /examples/chapter012/12_1_5/css/style.css

At first glance, nothing has changed in Figure 12.9 compared to Figure 12.6. But when you start scrolling the screen, the element set with `fixed` for the CSS feature `position` stays in place and doesn't scroll, as you can see in Figure 12.10. The HTML document to run can be found in */examples/chapter012/12_1_5/index.html*.



Article 2

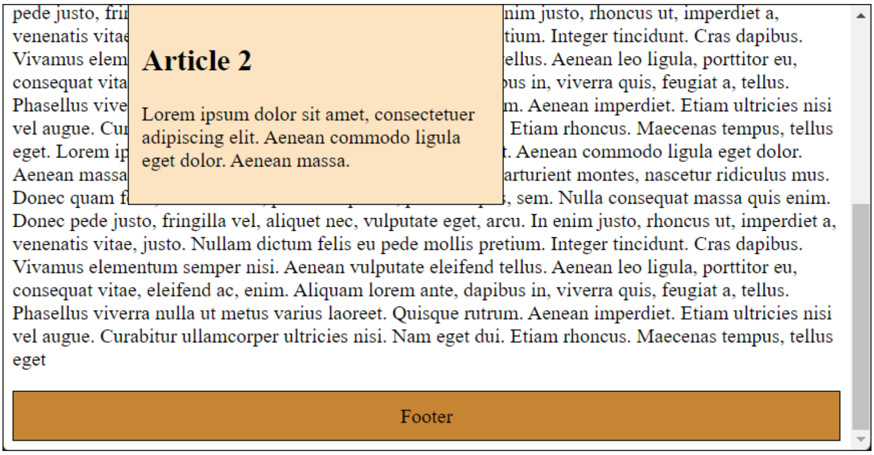
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Article

Lorem ipsun
adipiscing el
eget dolor. Aenean massa.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus.

Figure 12.9 At First, Everything Looks the Same with “position: fixed;”



Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

pede justo, fri nim justo, rhoncus ut, imperdiet a,
venenatis vita tium. Integer tincidunt. Cras dapibus.
Vivamus elem ellus. Aenean leo ligula, porttitor eu,
consequat vita us in, viverra quis, feugiat a, tellus.
Phasellus vive m. Aenean imperdiet. Etiam ultricies nisi
vel augue. Cu Etiam rhoncus. Maecenas tempus, tellus
eget. Lorem ip . Aenean commodo ligula eget dolor.
Aenean massa arturient montes, nascetur ridiculus mus.
Donec quam f , sem. Nulla consequat massa quis enim.
Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget

Footer

Figure 12.10 When You Start Scrolling the Web Page, the Difference Becomes Obvious Because the Element Won't Move

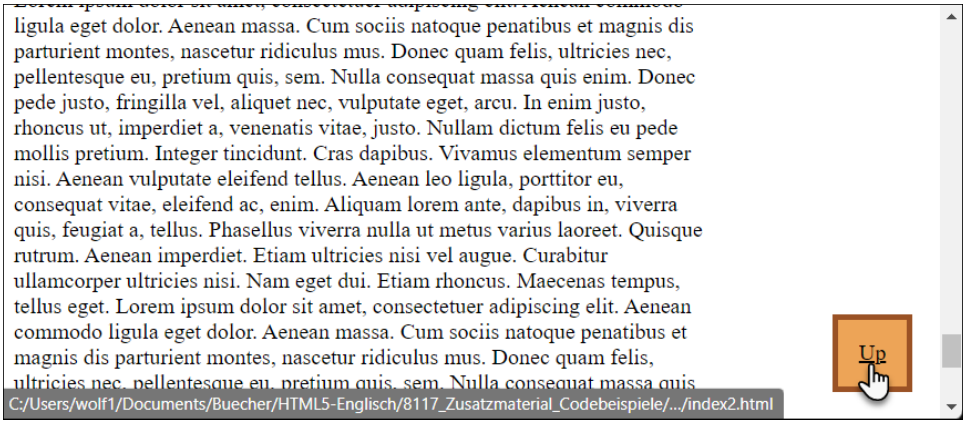
In practice, such fixed positioning is suitable, for example, for navigation areas, a fixed header or footer, or a fixed link to the top of the page that is always present at the same position. Here's a small example, where a link to take visitors back to the top of the document has been placed in a fixed position at the bottom right of the web page:

```
.up {
  position: fixed;
  bottom: 20px;
  right: 20px;
  padding: 10px;
  border: black 1px solid;
  background-color: orange;
}
```

Listing 12.8 /examples/chapter012/12_1_5/css/style.css

```
...
  <body>
      <h1 id="start">Start of page</h1>
      <a href="#start" class="up">Up</a>
      <p> ... </p>
      <p> ... </p>
      <p> ... </p>
  </body>
...
```

Listing 12.9 /examples/chapter012/12_1_5/index2.html



ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/.../index2.html

Figure 12.11 With the Fixed Positioning of the "Up" Link at the Bottom Right, You Can Jump Up to the Top of the Page at Any Time

12.1.6 Sticky Positioning (“position: sticky”)

Sticky positioning is a mixture of relative and fixed positioning. When you load the web page, the element behaves like an ordinary element. However, if you use it to hit the edge of the screen when scrolling up or down, the element with `position: sticky;` will stick there and then behave as with fixed positioning. Let’s look at a simple example:

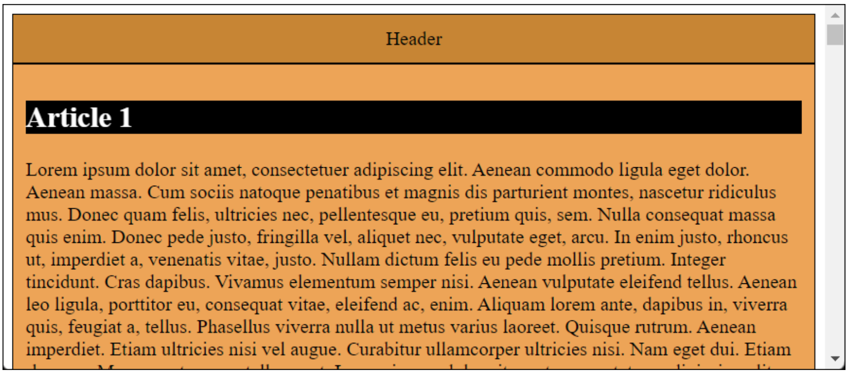
```
...
.sticky_h1 {
  position: -webkit-sticky;
  position: sticky;
  top: -1px;
  width: 100%
  background-color: black;
  color: white;
}
```

Listing 12.10 /examples/chapter012/12_1_6/css/style.css

```
...
  <body>
    <header class="foothead">Header</header>
      <article class="article01">
        <h1 class="sticky_h1">Article 1</h1>
        <p>Lorem ipsum dolor sit amet ... </p>
      </article>
      <article class="article02">
        <h1 class="sticky_h1">Article 2</h1>
        <p>Lorem ipsum dolor sit amet ... </p>
      </article>
    <footer class="foothead">footer</footer>
  </body>
...
```

Listing 12.11 /examples/chapter012/12_1_6/index.html

Here, a `sticky_h1` class was created for the article headings in an HTML document. The CSS feature `position` was passed the value `sticky`. The actual positioning (here, `top: -1px`) applies once the top of the screen is reached when scrolling down. In the example, you’ll find two longer articles. If you scroll down during this process, the heading will stick to the top of the screen (see Figure 12.13). This continues until you come across the next heading, which in turn makes it stick to the top (see Figure 12.14). This way, you always have the headline of the current article in view. If you scroll the web page all the way up again, this fixation comes off again.

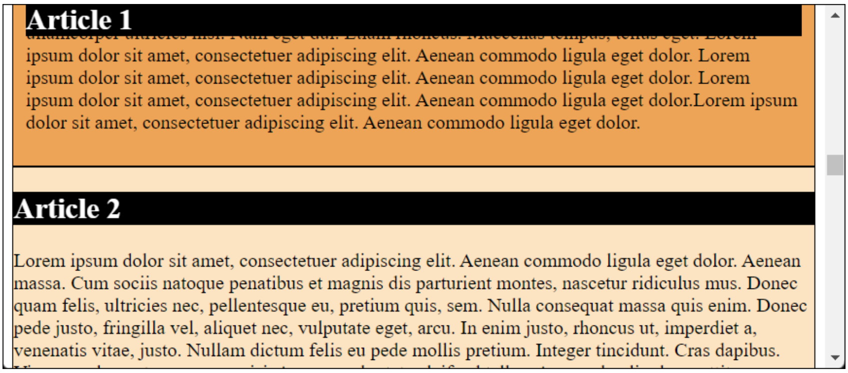


Header

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam

Figure 12.12 The Web Page after Loading: The Headline Is Placed as Usual



Article 1

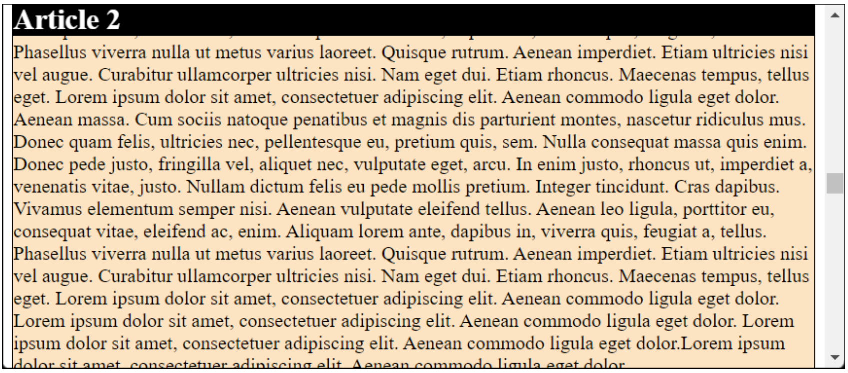
ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus.

Figure 12.13 When Scrolling Down, the Headline Will Stick to the Top of the Screen Due to "position: sticky;"

A look at *https://caniuse.com/#feat=css-sticky* shows that at the time this book went into print, support is still beset with some minor problems. However, all modern web browsers can already handle it quite well.



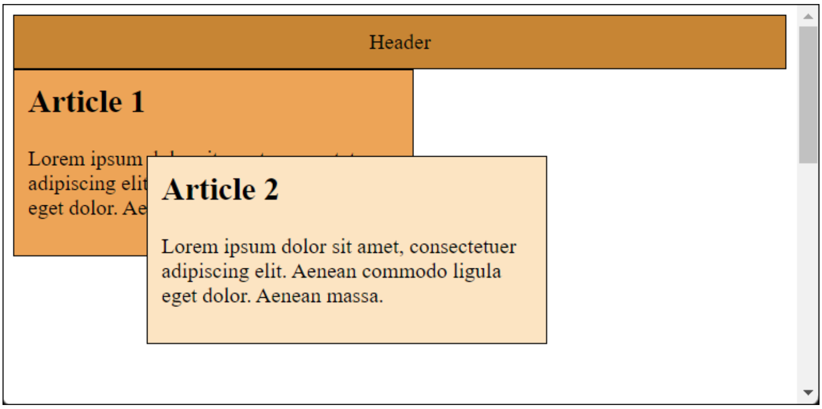
Article 2

Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor.Lorem ipsum

Figure 12.14 The Heading Will Stick until It Encounters Another Heading for Which "position" Also Equals "sticky"

12.2 Controlling Stacking Using “z-index”

In the previous examples with relative and absolute positioning, you’ve already seen that overlapping of elements can occur if you change elements absolute (or relative or fixed) in their position or remove them from the document flow. You can see this again in Figure 12.15.



Header

Article 1

Lorem ipsum
adipiscing elit
eget dolor. Ae

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Figure 12.15 With Relative or Absolute (or Even Fixed) Positioning, You Must Expect Elements to Overlap

By default, the relative, absolute, and fixed elements are stacked in the order they were written in the document flow of the HTML document. The last element written is on top. You can change this behavior using the CSS feature `z-index`. The CSS feature `z-index` is effectively the third axis for an element. So far, you’ve only taken care of the horizontal and vertical position (i.e., the x- and y-axis). The third dimension is the z-axis, on which the elements are superimposed, as Figure 12.16 shows more clearly.

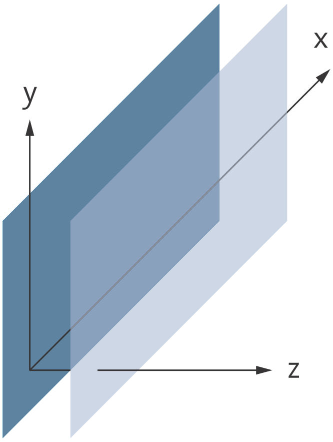


Figure 12.16 Elements Whose CSS Feature “position” Differs from the Default Value “static” Contain a Z-Axis in Addition to the X- and Y-Axis

Using the CSS feature `z-index` is very simple. The higher the noted value of `z-index`, the higher the element is in the stack. The element with the highest `z-index` value is thus at the top, and the element with the lowest `z-index` value is at the bottom. In case of equal values for `z-index`, the element that was last written in the document flow is on top.

As you can see in Figure 12.15, the CSS code still looks as follows:

```
...
      .article01 {
        width: 300px;
        padding: 10px;
        border: 1px solid black;
        background-color: sandybrown;
      }
      .article02 {
        position: relative;
        top: -75px;
        left: 100px;
        width: 300px;
        padding: 10px;
        border: 1px solid black;
        background-color: bisque;
      }
...
```

Listing 12.12 /examples/chapter012/12_2/css/style.css

```
...
      <article class="article01">
        <h1>Article 1</h1>
        <p>Lorem ipsum dolor ... </p>
      </article>
      <article class="article02">
        <h1>Article 2</h1>
        <p>Lorem ipsum dolor ... </p>
      </article>
...
```

Listing 12.13 /examples/chapter012/12_2/index.html

In this example, the second `article` element, which was moved relative to the `article02` class selector, overlapped the first `article` element.

If you now want the first `article` element to be placed on top of the second `article` element, you can use the CSS feature `z-index` in class selector `article02`. If you've used the CSS feature `z-index` in the `article02` class selector with any positive value that is lower than the `z-index` value in the `article01` class selector, you might be confused at first to find that nothing has changed. This is because stacking or using the CSS feature `z-index` has no effect on statically positioned elements, as is the case in the example with the first `article` element and the `article01` class selector.

In our case, you can solve the problem by providing the first article with the CSS feature `position: relative;` without using any values for positioning. The element will then remain in place as before. In the surrounding element, however, the element is considered to be positioned. So here’s the solution to the problem, based on which the first `article` element overlays the second `article` element using the CSS feature `z-index`:

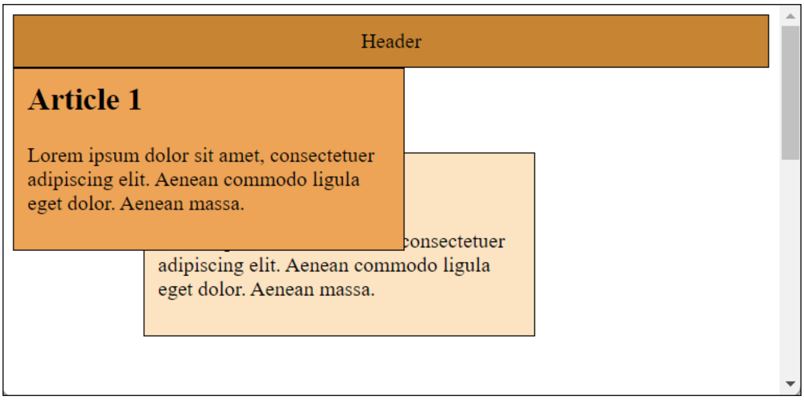
```
...
.article01 {
  position: relative;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: orange;
  z-index: 2;
}
.article02 {
  position: relative;
  top: -75px;
  left: 100px;
  width: 300px;
  padding: 10px;
  border: 1px solid black;
  background-color: yellow;
  z-index: 1;
}
...
```

Listing 12.14 /examples/chapter012/12_2/css/style2.css

```
...
      <article class="article01">
        <h1>Article 1</h1>
        <p>Lorem ipsum dolor ... </p>
      </article>
      <article class="article02">
        <h1>Article 2</h1>
        <p>Lorem ipsum dolor ... </p>
      </article>
...
```

Listing 12.15 /examples/chapter012/12_2/index2.html

In Figure 12.17 (as opposed to Figure 12.15), you can see how the first `article` element is placed over the second `article` element in the stack. This is due to the higher `z-index` value (here, `2`), which is used in the first `article` element with class selector `article01`. The second `z-index` value (here, `1`) in class selector `article02` didn't necessarily have to be specified. But you're welcome to increase the value of the CSS feature `z-index` in class selector `article02` to `3` in order to see that the second `article` element will then be placed over the first one again.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

Figure 12.17 The CSS Feature "z-index" Can Be Used to Adjust the Order in the Stack of Relative, Absolute, and Fixed Positioned Elements

Negative Values for "z-index"

You can also use negative values for the CSS feature `z-index`. Of course, the rule here is still that elements with positive `z-index` values get positioned above the elements with negative `z-index` values.

12.3 Floating Boxes for Positioning via "float"

Another important CSS feature for creating a layout with CSS is `float`. This feature enables you to take an element out of the usual document flow and place it on the right or left edge of the embracing element. The subsequent elements (without `float`) flow around this floated element. The CSS feature `float` was used for years to create layouts for websites. You can certainly still use this technique today, but there are now better techniques available for this, such as flexboxes or the grid layout. The classic example, which is still often used today to demonstrate the CSS feature `float` in use, is the flowing of text around an image. Let's first take a look at the following HTML lines:

```
...
  <body>
    <h1>A report</h1>
    <figure>
        <img src="images/image01.jpg" alt="Placeholder">
        <figcaption>An image</figcaption>
    </figure>    <p>Lorem ipsum dolor ... </p>
    <p>Lorem ipsum dolor ... </p>
    <p>Lorem ipsum dolor ... </p>
  </body>
...
```

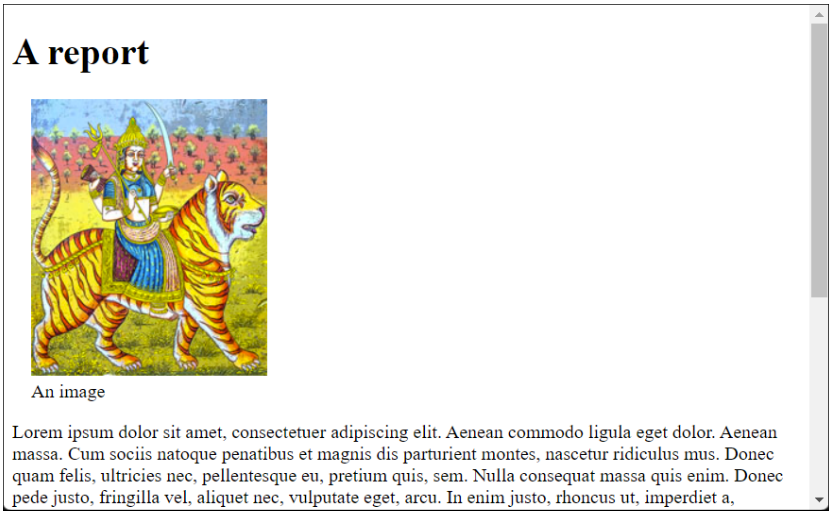
Listing 12.16 examples/chapter012/12_3/index.html

As befits a standard static positioning, the individual elements are arranged one below the other, as you can see in Figure 12.18.

If you apply the CSS feature `float` with the value `left` or `right` to the image, in this case inside the `figure` element, the text will flow around the image. You can do this via a simple type selector:

```
...
figure {
  float: left;
  margin: 0 1rem 0 0;
}
...
```

Listing 12.17 /examples/chapter012/12_3/css/style.css



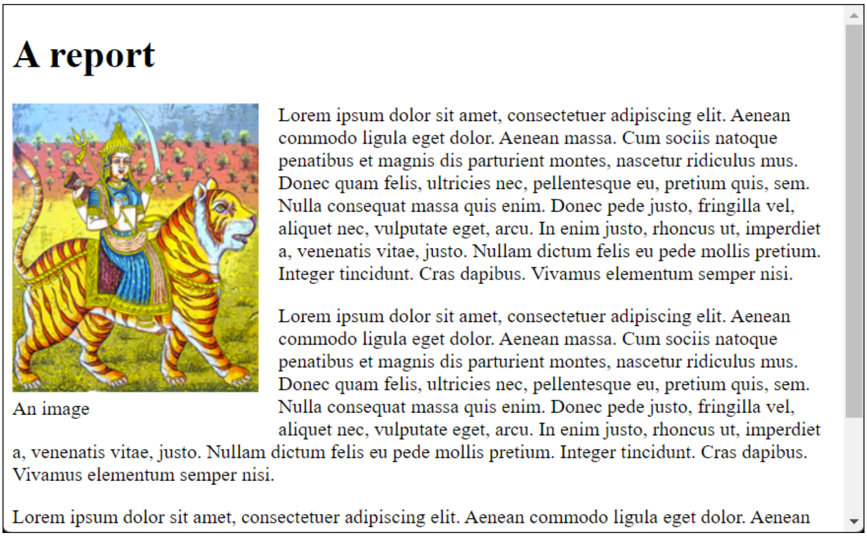
A report

An image

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a,

Figure 12.18 The Typical Document Flow with Standard Positioning

This will cause the `figure` element to float to the left along with the image and caption, while the subsequent elements, here the paragraphs with the `p` element, flow around the image (see Figure 12.19). The `margin` feature was only used here so that the text won't be "glued" too tightly to the image and there's a small buffer in between.



A report

An image

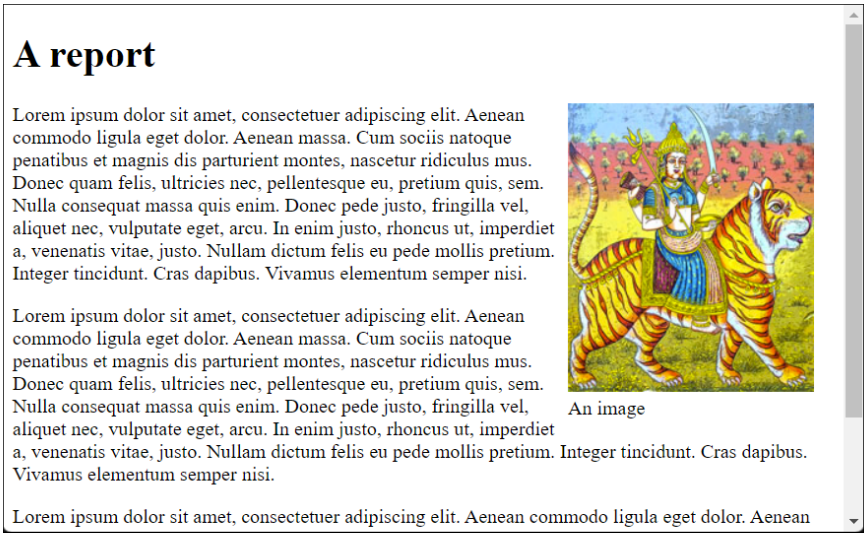
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean

Figure 12.19 The Image Was Floated with "float: left" on the Left, While the Following Paragraphs with the Text Flow around the Image

Besides `left` and `right`, you can use the values `none` (default) and `inherit` for `float`. `none` allows you to specify that the element shouldn't be floated. With the `inherit` value, on the other hand, the element `inherits` the `float` value of the parent element in which it resides.



A report

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

An image

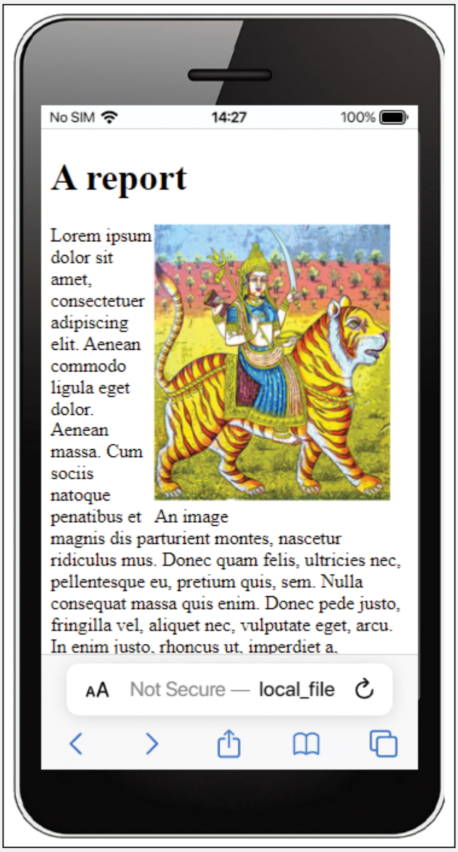
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean

Figure 12.20 Here, the <figure> Element Has Been Set to the Value "right" Using "float", and Consequently the Image It Contains Is Right-Aligned

Floating elements work only horizontally. The elements can only flow around to the left or to the right, whereas upward or downward movements aren’t possible. In addition, only the elements *after* a floated element flow around it. Elements that were written before the floated element in the document flow won’t be affected.

Testing a Narrow Viewport

If you use floated boxes, you shouldn’t forget to test the result on a narrower viewport such as a smartphone, especially to make sure that the text is still present next to the image and not displayed as one word per line. That doesn’t look nice, as you can see in Figure 12.21. Here, it would be a good idea to use a smaller image, or you could make the image responsive (which I’ll describe in Chapter 13, Section 13.3.2).



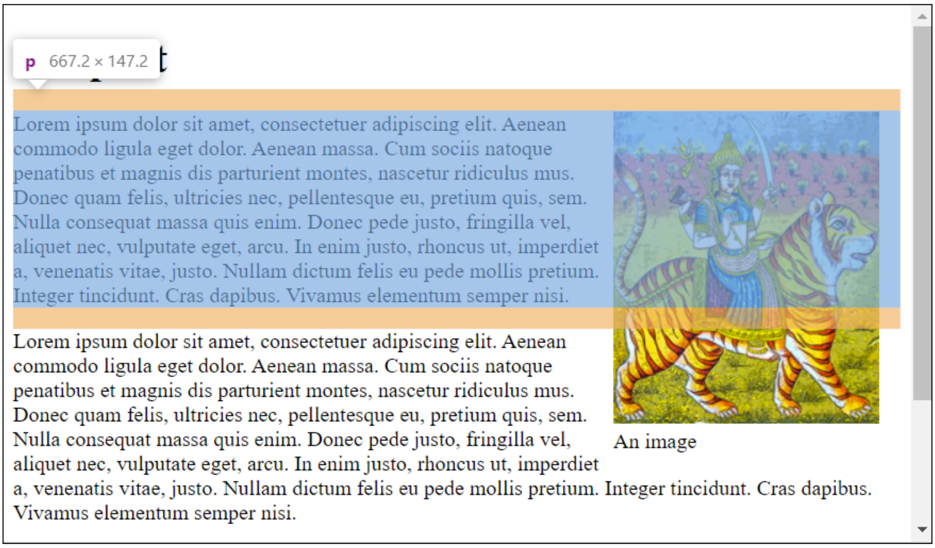
A report

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a,

An image

Figure 12.21 Layout No Longer Looks Nice on a 320-Pixel-Wide Smartphone: In Some Places, There’s Only One Word Left in the Line

At this point, it’s still important to know that only the text flows around the image, but not the `padding`, `border`, `margin`, and `background` features. They remain behind the floated image. You should know this if, for example, you want to use `margin` on the text to adjust the distance to the image and wonder why this won’t work.



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

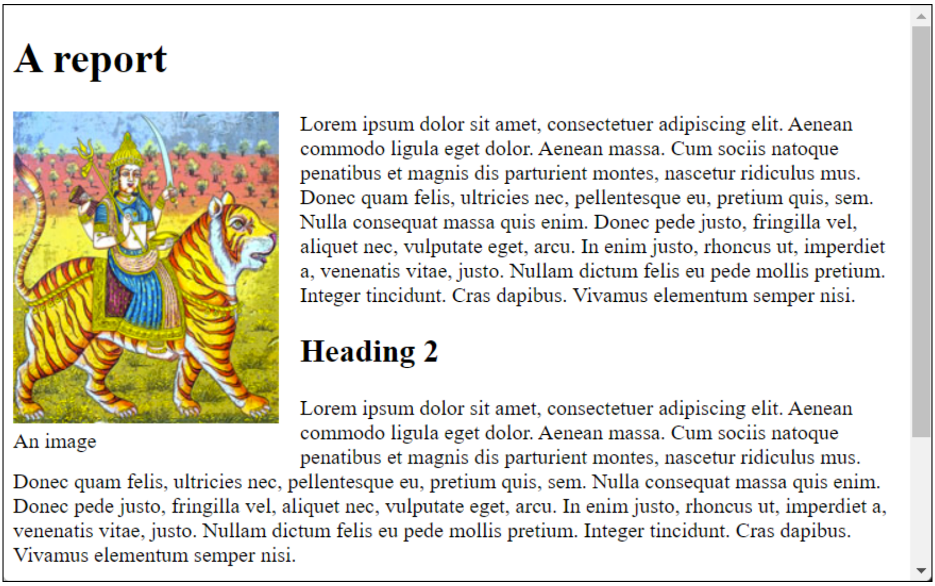
An image

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Figure 12.22 The Proof: Only the Text from the "p" Paragraph Element Flows around the "figure" Element with the Image; "padding", "border", "margin", and "background" Remain

12.3.1 Terminating the Float

You've certainly noticed in the example with the floated element that the next `p` element also flows around the image. This is convenient because then you don't have to worry about it, but that's not always desired. For example, if you want to start a new paragraph with a new heading in the next paragraph, this will look messy.



A report

An image

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Heading 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Figure 12.23 The Next Paragraph with the "h2" Heading Also Flows around the Image

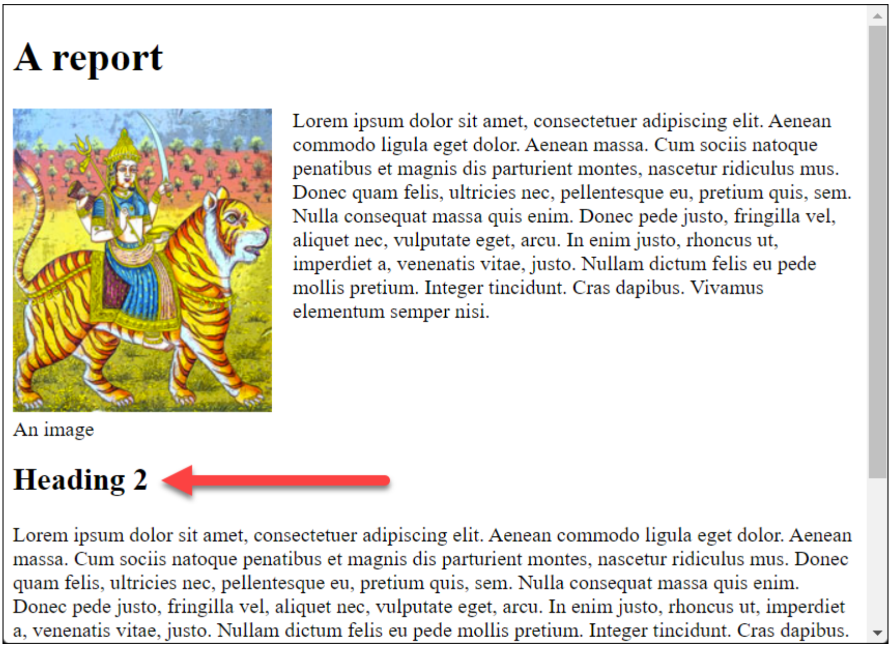
You can stop this floating around using the CSS feature `clear`. The `clear` feature allows you to disable the floating behavior for the subsequent elements. You can pass the values `left`, `right`, `both`, or `none` to the CSS feature `clear`. A `clear: left` ends a `float: left`, a `clear: right` ends a `float: right`, and a `clear: both` ends both. For this reason, it isn’t wrong to always use `clear: both`. The value `none` is the default value, which you can use if you want the elements to flow around each other again

In the following example, I’ll terminate the float using `clear: both;`. Because the image is floated around on the left, I could also have used `clear: left;`.

```
...
.float-left {
    float: left;
    margin: 0 1rem 1rem 0;
}
.endfloat {
    clear: both;
}
...
```

Listing 12.18 /examples/chapter012/12_3/css/style2.css

In the sample document, I include the class to stop reflowing in the `h2` heading, which stops reflowing exactly at that point and for all subsequent elements, as you can see in Figure 12.24.



A report

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

An image

Heading 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus.

Figure 12.24 From the h2 Heading Onward, the Flow around the Image Will End

Depending on the screen width, however, a gap opens up here.

```
...
    <h1>A report</h1>
    <figure class="float-left">
        <img src="images/image01.jpg" alt="Placeholder">
        <figcaption>An image</figcaption>
    </figure>
    <p>Lorem ipsum dolor ... </p>
    <h2 class="endfloat">Heading 2</h2>
    <p>Lorem ipsum dolor sit amet ... </p>
...
```

Listing 12.19 /examples/chapter012/12_3/index2.html

12.3.2 Combining Floats into One Entity

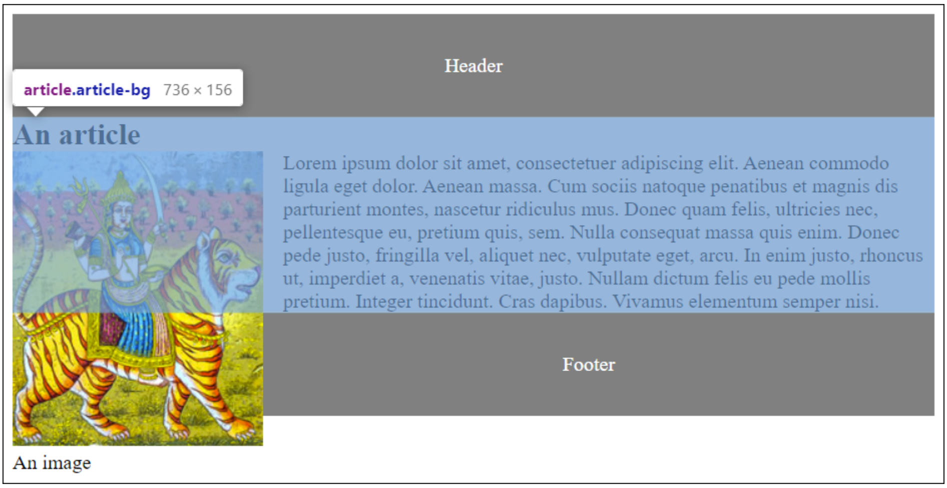
As already shown in Figure 12.22, only the text flows around the image, and the `padding`, `border`, `margin`, and `background` features remain behind the floated image in the document flow. In the example, the image also protrudes from the parent element. Because further text follows behind it in the example, that didn't bother me any further. It also isn't recommended at first if you put everything into a parent element, such as the following:

```
...
    <header class="head-foot">Header</header>.
    <article class="article-bg">
        <h1>An article</h1>
        <figure class="float-left">
            <img src="images/image01.jpg" alt="Placeholder">
            <figcaption>An image</figcaption>
        </figure>
        <p>Lorem ipsum dolor sit ... </p>
    </article>
    <footer class="head-foot">Footer</footer>
...
```

Listing 12.20 /examples/chapter012/12_3/index3.html

A `clear: left;` with the `footer` would fix the problem for now, but the problem remains when styling the `padding`, `border`, `margin`, and `background` features, as shown in Figure 12.26. There, a gray background is used for the `article` element for the purpose of clarity.

There's a solution to this with `display: flow-root;`, as this creates a new block for the surrounding element via CSS. If you use `display: flow-root;` for the `article` element, the elements it contains are enclosed in a block, resulting in Figure 12.27. Now the thing is that `display: float-root;` is available to all newer web browsers but not to older browsers. For this purpose, the ancient trick with `overflow: hidden;` is useful, which also encloses the content in a new block as a side effect.



Header

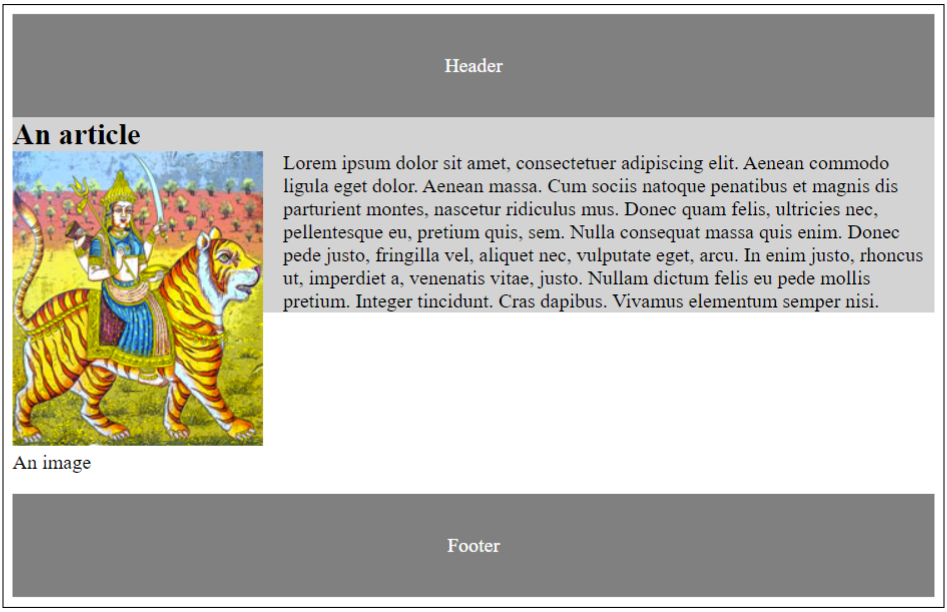
An article

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Footer

An image

Figure 12.25 The Image Extends from the “article” Element beyond the “footer” Element



Header

An article

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

An image

Footer

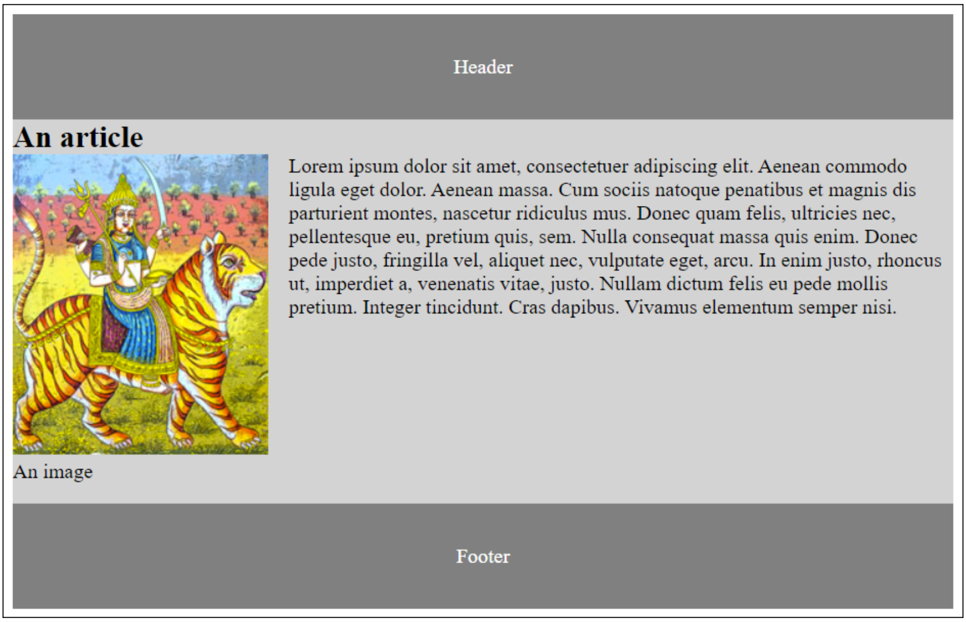
Figure 12.26 Stopping the Float Solves Only Part of the Problem: With the CSS Features “padding”, “border”, “margin”, and “background”, the Image Remains Protruding

An elegant solution to this is the CSS feature query `@supports()`. It allows you to check whether a browser can handle certain CSS property-value combinations. With regard to our example, this could be implemented as follows:

```
...
    .article-bg {
        background-color: lightgray;
        overflow: hidden;
    }

    @supports(display: flow-root) {
        .article-bg {
            display: flow-root;
            overflow: initial;
            background-color: lightgray;
        }
    }
...
```

Listing 12.21 /examples/chapter012/12_3/css/style3.css

First, you set the `overflow: hidden;` feature in the `article-bg` class. If the browser doesn't know the `@supports()` query, then it simply continues. A browser that knows the `@supports()` query checks whether it can handle `display: flow-root`, and, if so, it will use the features inside the statement block where `display: flow-root;` is set and `overflow` is provided with `initial`.



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi.

Figure 12.27 Now the Floated "figure" Element inside the "article" Element Has Been Combined into a New Block with the "p" Element

12.4 Flexible Boxes of CSS

As the topic of *responsive web design* is becoming increasingly important, the desire for a simpler and better alternative to positioning according to the `float` principle is getting stronger as well. One of these alternatives is the extremely promising flexbox model. The CSS flexbox is perfect for arranging elements next to or below each other. This is very useful for galleries or links of a navigation, for example, because the CSS flexboxes provide even more options, such as neatly arranging the elements next to each other with a certain spacing, in a certain order, or in a certain size.

The principle of flexboxes is simple and can be quickly explained: You need a parent element in which you set the CSS feature `display` to `flex`. This feature affects all child elements contained in it. The parent element that has been given the CSS feature `display: flex` is also referred to as a *flex container*. The child items contained in it are the *flex items*.

12.4.1 Aligning the Flexbox

The CSS feature `flex-direction` allows you to specify how to align the elements within the flexbox. For a horizontal alignment, you can use the `row` value, and for a vertical alignment, you can use the `column` value. If you don't use the CSS feature `flex-direction`, then `row` is the default setting. Thus, the default orientation for the elements of a flexbox is horizontal.

Here's an example that demonstrates these basic features of flexboxes in use:

```
.myarticle {
  width: 300px;
  padding: 10px;
  margin: 0px 5px 5px 0px;
  border: 1px solid black;
  background-color: bisque;
}
.mymain {
  width: 90%;
  padding: 10px;
  background-color: sienna;

  display: flex;
  flex-direction: row;
}
```

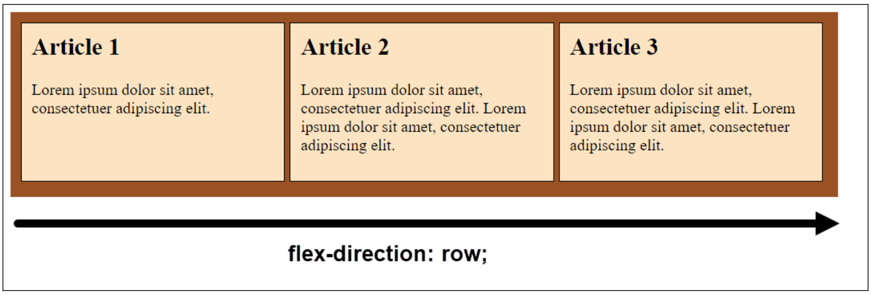
Listing 12.22 /examples/chapter012/12_4_1/css/style.css

```
...
    <main class="mymain">
```

```
      <article class="myarticle">
        <h1>Article 1</h1>
        <p>Lorem ipsum dolor sit amet ...</p>
      </article>
      <article class="myarticle">
        <h1>Article 2</h1>
        <p>Lorem ipsum dolor sit amet ...</p>
      </article>
      <article class="myarticle">
        <h1>Article 3</h1>
        <p>Lorem ipsum dolor sit amet ... </p>
      </article>
    </main>
...
```

Listing 12.23 /examples/chapter012/12_4_1/index.html

Here, only the `mymain` class selector was used to set the `display` type of the parent element `<main>` to `flex` and the alignment of `flex-direction` to `row`. You can also omit the specification with `flex-direction` because `display: flex` is set up for this setting by default. These two specifications cause all child elements contained in the `main` element (here again, `article` elements) to be aligned horizontally in the flexbox, which have been colored in gray here for clarity, as you can see in Figure 12.28.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

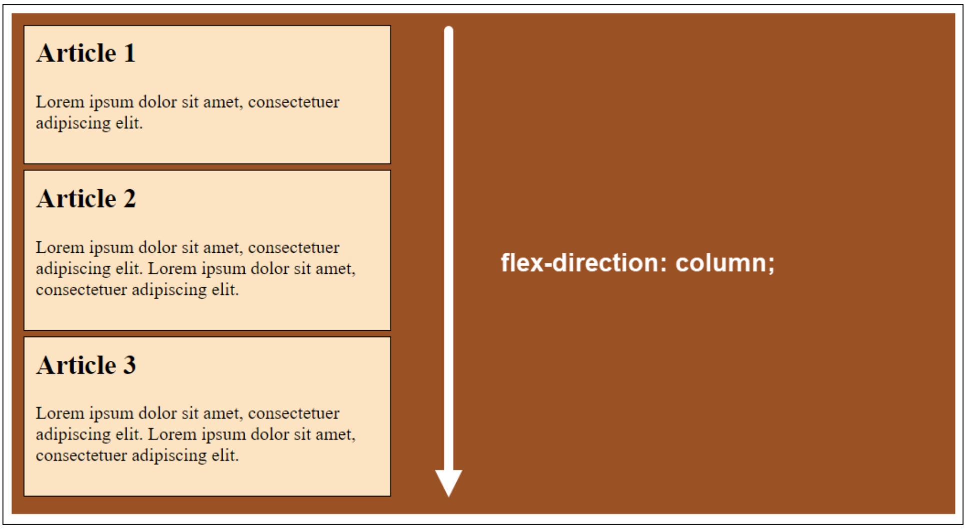
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.28 Flexbox in Horizontal Direction

If you use the `column` value for the CSS feature `flex-direction` instead, the individual elements within the `main` element will be vertically aligned, which is shown in Figure 12.29.

Sorting in Reverse Order

For the CSS feature `flex-direction`, you can use the values `row-reverse` and `column-reverse`, which will sort and display the contained elements in reverse order. With regard to our example, **Article 3** would be displayed first, then **Article 2**, and finally **Article 1**. Just try these values out for yourself.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

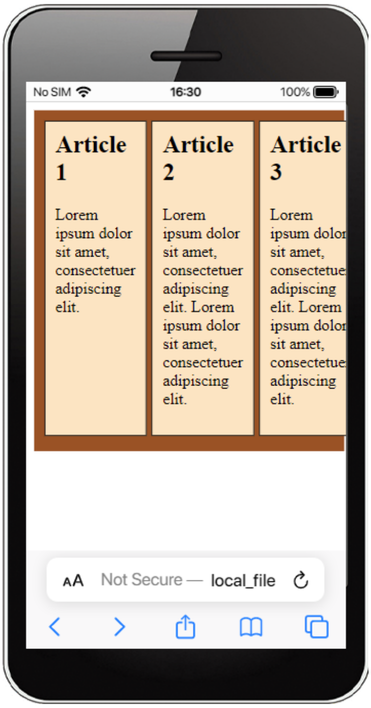
Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.29 Flexbox in Vertical Orientation (/example/chapter012/12_4_1/index2.html)

Wrapping Elements in a Flexbox: “flex-wrap”

The unattractive aspect about the example in */examples/chapter012/12_4_1/index.html* is that it doesn’t look nice beyond a certain window width, and the elements end up flowing beyond the surrounding flexbox, as you can see in Figure 12.30.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolo sit amet, consectetue adipiscing elit. Lorem ipsum dolo sit amet, consectetue adipiscing elit.

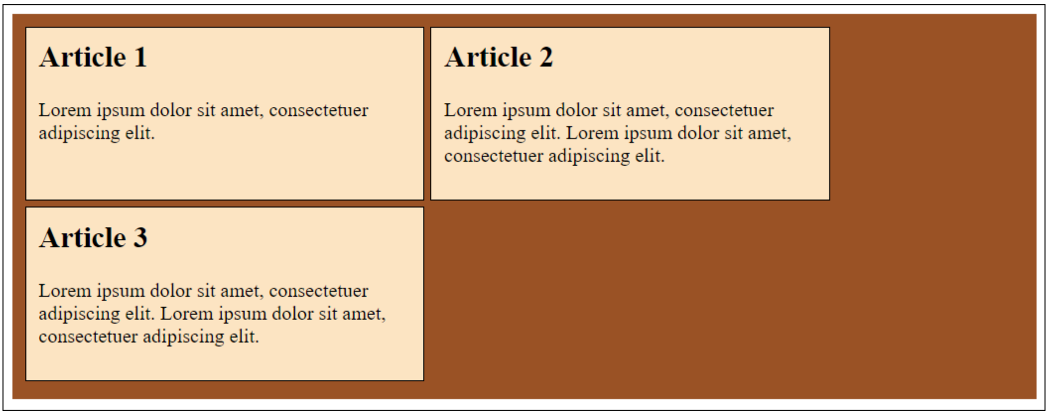
Figure 12.30 At Some Point, the Flexibility of a Flexbox Also Comes to an End

If you want a behavior where the elements wrap to the next row, the flexbox model provides the CSS feature `flex-wrap` for this purpose. The default value `nowrap` prevents the elements in the flexbox from wrapping. If you use the `wrap` value for this, the elements wrap into a new row. Besides `nowrap` and `wrap` for the CSS feature `flex-wrap`, there's the `wrap-reverse` value, which you can use to wrap flexible elements to the top.

Following is an example of `flex-wrap`:

```
...
.mymain {
  width: 95%;
  padding: 10px;
  background-color: sienna;
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}
```

Listing 12.24 /examples/chapter012/12_4_1/css/style3.css



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.31 Thanks to “flex-wrap: wrap;” the Elements in a Flexbox Wrap into a New Row (/examples/chapter012/12_4_1/index3.html)

“flex-flow”, the Short Notation for “flex-direction” and “flex-wrap”

With `flex-flow` there’s a short notation available for `flex-direction` and `flex-wrap`. It doesn’t matter in which order you write the values. Let's look at an example specification:

```
flex-flow: row wrap;
```

This corresponds to the following notation:

```
flex-direction: row;
flex-wrap: wrap;
```

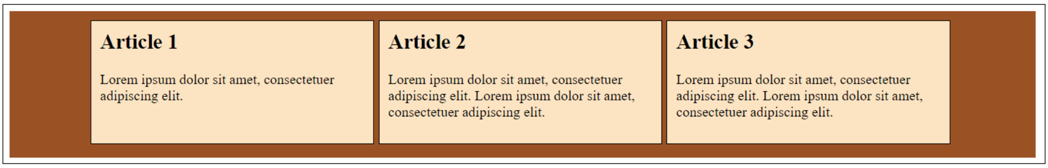
Arranging Elements along the Main Axis: "justify-content"

With reference to Figure 12.28 and the example in */examples/chapter012/12_4_1/index.html*, you can arrange the individual elements with the CSS feature `justify-content`. Possible values for this are `flex-start`, `center`, `space-between`, and `space-around`, for example:

```
...
.mymain {
  ...
  display: flex;
  justify-content: center;
}
```

Listing 12.25 /examples/chapter012/12_4_1/css/style4.css

As you can see in Figure 12.32 for the example in */examples/chapter012/12_4_1/index4.html*, `justify-content: center;` centers all child elements within class selector `mymain`.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

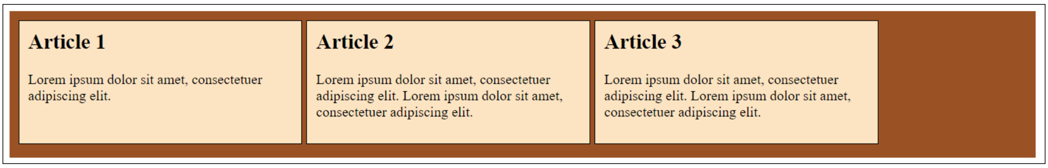
Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.32 You Can Use "justify-content: center;" to Center the Elements



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

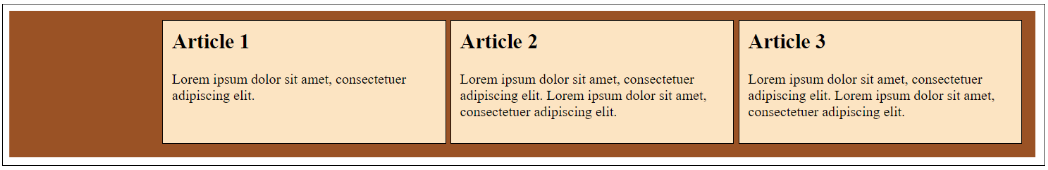
Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.33 "justify-content: flex-start;" Allows You to Arrange the Elements Left-Justified



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

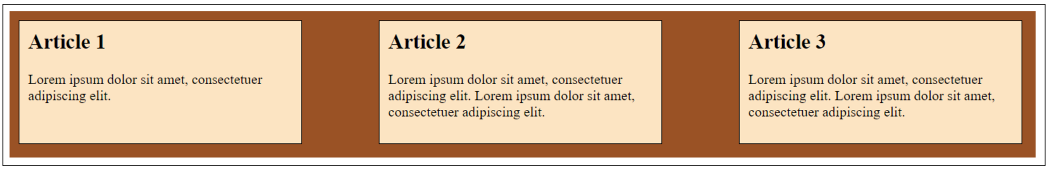
Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.34 "justify-content: flex-end;" Enables You to Arrange the Elements Right-Justified



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

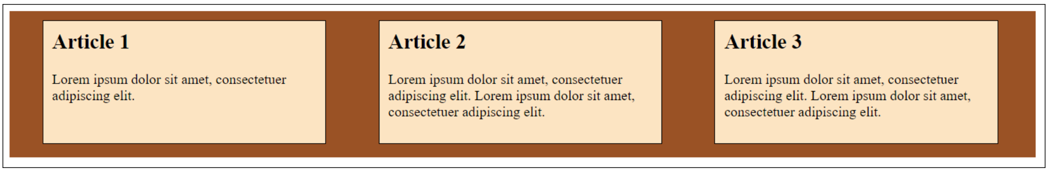
Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.35 "justify-content: space-between;" Makes Sure That the Elements Are Arranged with Equal Spaces In Between: The First and Last Elements Are Located at the Beginning and End of the Line, Respectively



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.36 "justify-content: space-around;" Ensures That All Elements Are Distributed Evenly

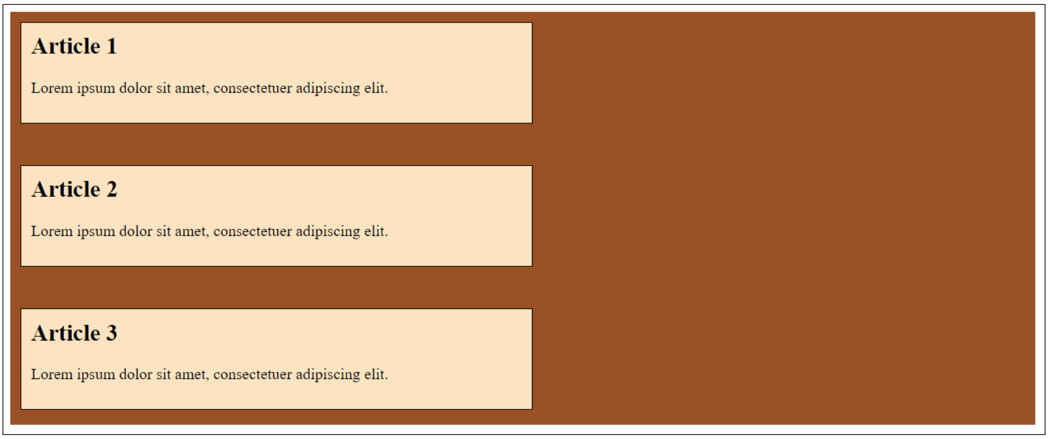
Arranging Elements along the Cross Axis: "align-content"

If you want to arrange elements along the cross axis, you can use the CSS feature `align-content`. The default value here is `stretch`. As you can see in Figure 12.29, this distributes all elements evenly. The other possible values for `align-content` are `flex-start`, `flex-end`, `center`, `space-between`, and `space-around`. The meaning of these values is the same as for `justify-content`, only for the cross-axis. For example, `flex-start` places the elements at the top, `flex-end` at the bottom, and `center` in the middle. `space-between` ensures an even distribution, with the first element at the top and the last at the bottom. `space-around`, on the other hand, distributes all elements evenly without treating the first or last element separately. Consider this example.

```
.myarticle {
  width: 500px;
  padding: 10px;
  margin: 0px 5px 5px 0px;
  border: 1px solid black;
  background-color: bisque;
}
.mymain {
  width: 95%;
  height: 500px;
  padding: 10px;
  background-color: sienna;
  display: flex;
  flex-wrap: wrap;
  align-content: space-between;
}
```

Listing 12.26 /examples/chapter012/12_4_1/css/style5.css

You can see the example from */examples/chapter012/12_4_1/index5.html* in Figure 12.37 during execution. If you want to see the other values `flex-start`, `flex-end`, or `space-around` in use, you just need to change the `align-content` value in the */examples/chapter012/12_4_1/css/style5.css* example accordingly.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.37 With "align-content: space-between;", the Elements Are Evenly Distributed: The First and Last Elements Are at the Top and Bottom, Respectively

Arranging Individual Elements Differently: "align-self"

If you want to assign a different property to individual elements in the arrangement of flexible elements than the one specified in the parent element, you can use the CSS feature `align-self` for this purpose. As a default value (`auto`), the value is taken from the parent element. Otherwise, you'll also find the values `stretch` (evenly distribute), `center` (center), `flex-start` (top), and `flex-end` (bottom) with the same result in the arrangement as I described it for `align-content`. In addition, you'll find `baseline` as a possible value, which you can use to align an element to the baseline.

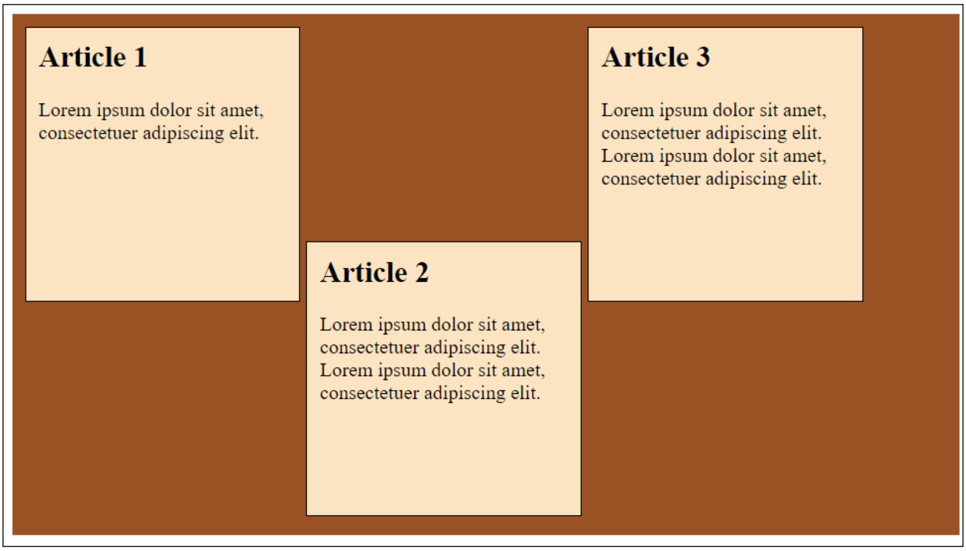
Let's look at a simple example: You are invited again to experiment with the values of `align-self`:

```
.myarticle {
  ...
}
.mymain {
  ...
  display: flex;
  flex-wrap: wrap;
  justify-content: flex-start;
}
.bottom {
  align-self: flex-end;
}
```

Listing 12.27 /examples/chapter012/12_4_1/css/style6.css

```
...
    <main class="mymain">
      <article class="myarticle">
        <h1>Article 1</h1>
        <p>Lorem ipsum ... </p>
      </article>
      <article class="myarticle  bottom">
        <h1>Article 2</h1>
        <p>Lorem ipsum ... </p>
      </article>
      <article class="myarticle">
        <h1>Article 3</h1>
        <p>Lorem ipsum ... </p>
      </article>
    </main>
...
```

Listing 12.28 /examples/chapter012/12_4_1/index6.html



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.38 Here I've Arranged the Middle Article with "align-self: flex-end;" at the Bottom of the Flexbox

12.4.2 Setting the Flexibility of the Flexbox

To set the flexibility of the corresponding elements within the flexbox, you can use the CSS feature `flex`. The property expects a numerical value. The numbers behave relatively, which means that an element with the specification `flex: 4` is four times as flexible as an element with the `flex` property: `1`.

Short Notation for "flex-grow", "flex-shrink", and "flex-basis"

The CSS feature `flex` is a short notation for the other existing CSS features of flexboxes—`flex-grow`, `flex-shrink`, and `flex-base`. Strictly speaking, the specification corresponds to `flex: 2` of the specification `flex-grow: 2`. You can also use the other two values for `flex-shrink` and `flex-base`. For example, take the following:

```
flex: 2 1 30%; /* flex-grow=2 flex-shrink=1 flex-basis=30% */
```

This specification is a short notation for the following:

```
flex-grow: 2;
flex-shrink: 1;
flex-basis: 30%;
```

Using `flex-grow`, you can control how flexibly the element grows relative to the rest of the elements (when zoomed in). You can specify how far the element shrinks relative to the other elements (when shrinking) using `flex-shrink`, and you can specify the basic width for the element via `flex-basis`. Besides percentages, you can use pixels (`px`), `em`, or other units. The default value for `flex-basis` is `auto`.

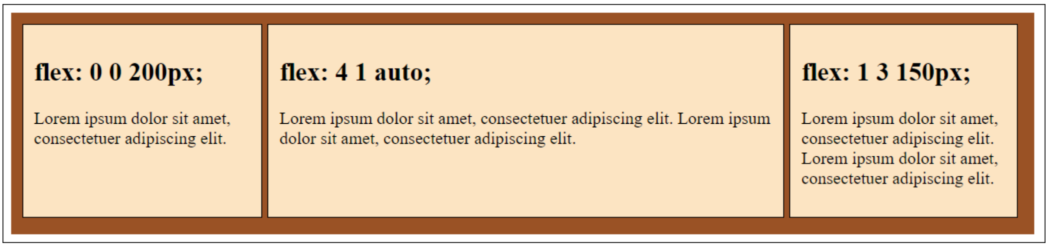
The default value of `flex` in general is `0 1 auto`(`flex: 0 1 auto`).

Here's an example that demonstrates the CSS feature `flex` in use:

```
...
.mymain {
  width: 90%;
  padding: 10px;
  background-color: sienna;

  display: flex;
}
.article01 { flex: 0 0 200px; }
.article02 { flex: 4 1 auto; }
.article03 { flex: 1 3 150px; }
...
```

Listing 12.29 /examples/chapter012/12_4_2/css/style.css

For the first `article` element with class selector `article01`, `flex: 0 0 200px;` causes the box to have zero flexibility relative to the other elements in the `main` element when zoomed in and out, respectively. The width of the box is initialized with 200 pixels and, because the two preceding values are `0`, can't change when the window is enlarged or reduced (see Figure 12.39 and Figure 12.40). The HTML document for this can be found in */examples/chapter012/12_4_2/index.html*.



flex: 0 0 200px;

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

flex: 4 1 auto;

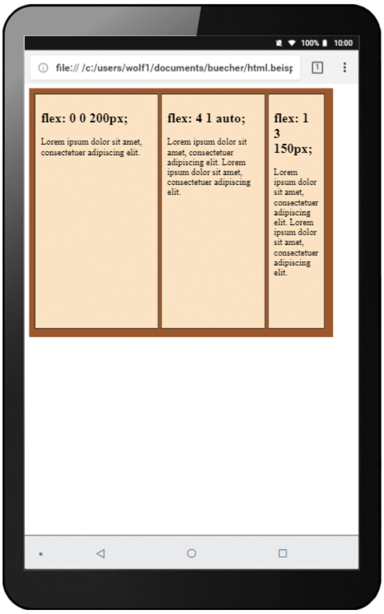
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

flex: 1 3 150px;

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.39 Different Values for Flexboxes

In the second `article` element with class selector `article02`, the box is four times more flexible than the other boxes when enlarged. When shrinking, the value was set to `1` and the basic width to `auto`. You could have omitted the `auto` value because it's the default value. In Figure 12.39, you can see that this item is always relatively larger than the other items when displayed wider. With a narrower viewport, as in Figure 12.40, this ratio applies only to the third article because the first article allows no flexibility and remains rigid at its 200 pixels.



flex: 0 0 200px;

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

flex: 4 1 auto;

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

flex: 1 3 150px;

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.40 Unlike Figure 12.39, a Small Device Was Used

In the last `article` element with class selector `article03`, the relative flexibility on zooming was set to `1`. When shrinking, on the other hand, this box is three times more flexible than the other boxes. The base value for the width was specified as 150 pixels. If you had used `auto` here, you would immediately have noticed this threefold flexibility when shrinking, which wouldn't look nice. In Figure 12.39, the base value of 150 pixels for the width still applies. Nevertheless, the article is already relatively smaller than the second article. For the smaller viewport in Figure 12.40, the effect of having three times the flexibility of the other two items is relatively clear.

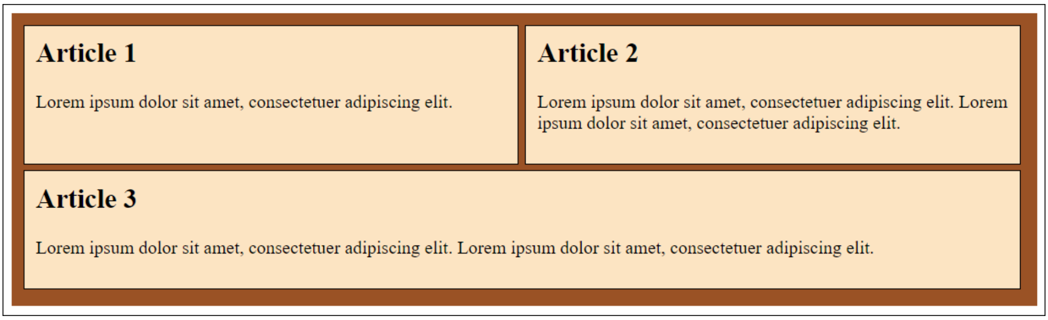
The Special Feature of "flex-grow" with Line Breaks

You'll also find a very interesting and useful feature if you allow a line break with `flex-flow: wrap` and set the `flex-grow` feature to `1`, for example:

```
...
.myarticle {
    width: 300px;
    ...
    flex-grow: 1;
}
.mymain {
    width: 95%;
    ...
    display: flex;
    flex-direction: row;
    flex-wrap: wrap;
}
...
```

Listing 12.30 /examples/chapter012/12_4_2/css/style2.css

If you run */examples/chapter012/12_4_2/index2.html* here and use a sufficiently large viewport, all `article` elements will be aligned side by side in the flexbox. If the viewport is now made smaller and the last `article` element slides down, the `article` element will take over the full width of the next line thanks to `flex-grow: 1`.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.41 If You Allow the Line Break and Use "flex-grow: 1", the Flex Item Wrapped to the Next Line Will Take the Complete Width of the Line

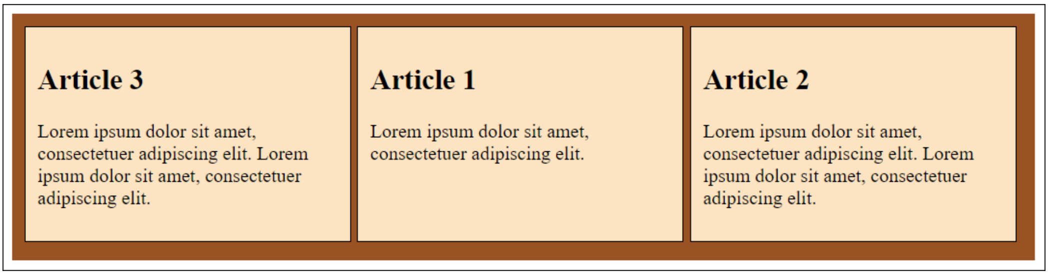
12.4.3 Determining the Order of the Boxes

Another very nice feature of flexboxes is that you can set the order yourself using the CSS feature `order`. Here, too, you must use a numerical value. Let's look at an example:

```
...
.article01 {  order: 2;  }
.article02 {  order: 3;  }
.article03 {  order: 1;  }
...
```

Listing 12.31 /examples/chapter012/12_4_3/css/style.css

Here, by `order: 1`, the third `article` element becomes the first; due to `order: 2`, the first `article` element becomes the second; and `order:3` declares the second `article` element to be the third. In Figure 12.42, you can see how the order has changed. The HTML document for this can be found in */examples/chapter012/12_4_3/index.html*.



Article 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Figure 12.42 You Can Change the Order of the Elements in the Container Element via the CSS Feature "order"

Additional Examples

When you've read this chapter, you'll know the basics of using flexboxes in practice. The main area of use for flexboxes is to arrange elements as neatly as possible next to or below each other. In practice, flexboxes are used for photo galleries, maps, or aligning form elements such as navigation or a contact form, among other things. Simple examples to study and test CSS flexboxes can be found at *http://quackit.com/css/flexbox/examples/*.

12.5 Summary

In this chapter, you learned a lot about positioning HTML elements with CSS. You also now know the following:

- How to position HTML elements statically, relatively, and absolutely using CSS feature `position`
- How to stack HTML elements that overlap in a relative or absolute positioning using the `z-index`
- How to remove HTML elements from the document flow using `float` and place them at the right or left edge of the embracing element, as well as how to remove the flow around the elements again via `clear`
- Which options are provided by the flexboxes in CSS, due to which the positioning of elements becomes almost a walk in the park

Chapter 13

Creating Responsive Layouts with CSS

When it comes to website layouts, the choice is likely to fall mostly on the responsive variants because here you no longer have to worry about the different screen sizes. This chapter provides an introduction on how to create responsive websites.

The way you should create your layouts isn't set in stone, nor is there a right or wrong here. Today, responsive layouts are used for this purpose, which deliver a web page that fits the width of the visitor's device. This chapter describes what responsive layouts are all about and how you can implement them.

Before we get started, here's an overview of what you'll learn in this chapter:

- The basic handling of media queries
- The fundamentals of what is important in responsive web design
- How to create a simple responsive layout
- How to use the CSS grid for responsive web design

To temper expectations a bit here, it's worth mentioning that you'll only learn the basics of what responsive layouts are and how you can use them in practice. The examples in this chapter are kept relatively simple. Responsive web design is a vast, ever-evolving subject, and there's a reason that entire books are devoted to it. Still, by the end of the chapter, you'll know what that's all about and how you can create responsive layouts.

13.1 Basic Theoretical Knowledge of Responsive Web Design

The way we access the internet today has become very versatile. Whereas a few years ago, a website was only viewed with a desktop PC or laptop, today many other devices such as tablets, smartphones, e-book readers, game consoles, or TV sets have joined the ranks. The challenge here is to respond to the screen size and screen resolution of each device with an appropriate layout.

Prior to the era of responsive web design, the appearance of a website was quite dependent on the device used to view it. Websites used to be optimized for the screen resolution of a desktop PC, but nowadays you have to rethink a little due to the rapidly increasing market share of smartphones and tablets. Standard smartphone resolutions start at 320–480 pixels, tablets are often 768–1,024 pixels, and common desktop PCs start at 1,024 pixels. Offering different mobile and desktop versions of a website is one way to solve the problem. However, that would take a considerable effort, and when a new tablet or smartphone format of the next generation is due, another version of the layout would become necessary.

Most of the time, the mobile-only versions were just stripped-down versions with reduced functionality of the desktop version, and they were swapped out to a subdomain with a single-column layout with *m* or *mobile* (e.g., *mobile.mydomain.com*). At the latest since the introduction of mobile devices such as smartphones and tablets, these slimmed-down mobile versions of a website are no longer satisfactory. In addition, the web browsers on mobile devices were and are technically on a very high level and at least equal to the desktop variants. It would therefore be a shame to give away the potential with a slimmed-down mobile version.

Mobile Dominates

Recent statistics confirm the trend that mobile devices are now the most used devices when visitors are on the web.

Instead of creating and maintaining countless layout versions for the same website, *responsive web design* or *responsive layout* is used. This technique takes into account the characteristics of the end device to adapt the website to achieve an optimal and user-friendly display for the end device. The main criteria for such a customized layout are the screen size (usually width) of the device and possibly the available input methods (mouse or touch screen).

The meaning of *responsive* is something like *reacting*. This sounds strange and is rather rarely used this way, but it sums things up quite well because with this technique, the structural design and content of a website reacts to the screen resolution of the users' devices, and the layout is output accordingly. So, when we talk about responsive layouts, the website adapts to the users' screens.

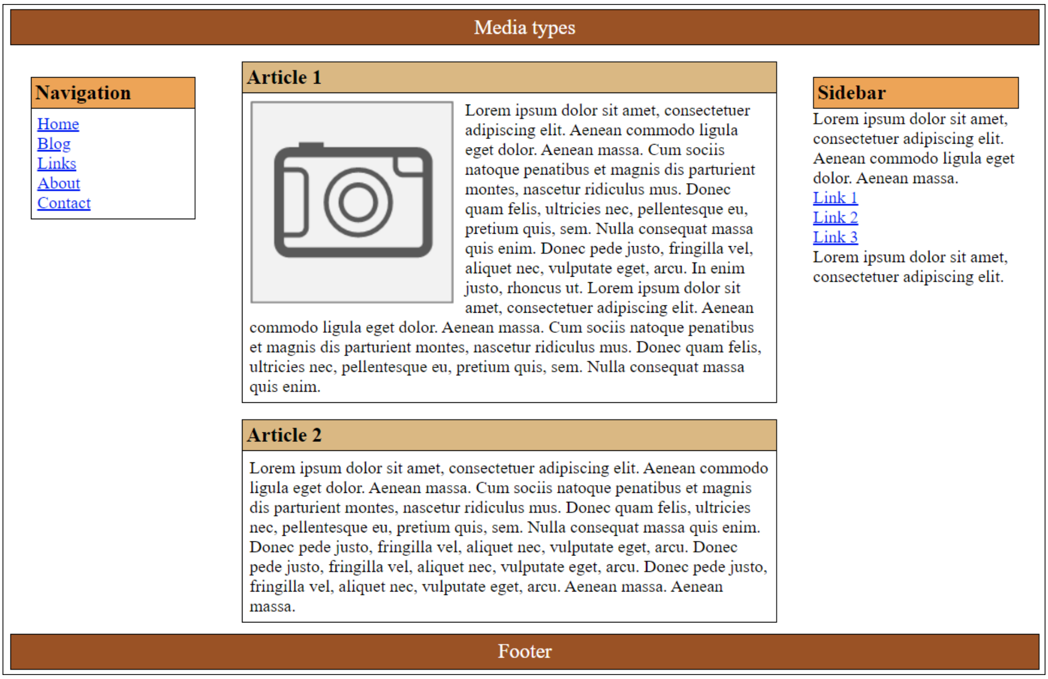
13.1.1 Using Specific Media Types

The idea of responding to specific media types has been around for a long time in CSS, which I covered briefly in Chapter 8, Section 8.3.8. In doing so, we've provided various separate stylesheets for the different output media such as the screen (`media="screen"`) or the printer (`media="print"`):

```
...
<link href="css/screen.css" rel="stylesheet" media="screen">
<link href="css/print.css" rel="stylesheet" media="print">
...
```

Listing 13.1 /examples/chapter013/13_1_1/index.html

Using the `link` element, you can provide a version for the screen (`media="screen"`) and a specific version for the printer. You can see the version for the screen with *screen.css* in Figure 13.1.



Media types

Navigation

- Home
- Blog
- Links
- About
- Contact

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. Aenean massa. Aenean massa.

Sidebar

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa.

- Link 1
- Link 2
- Link 3

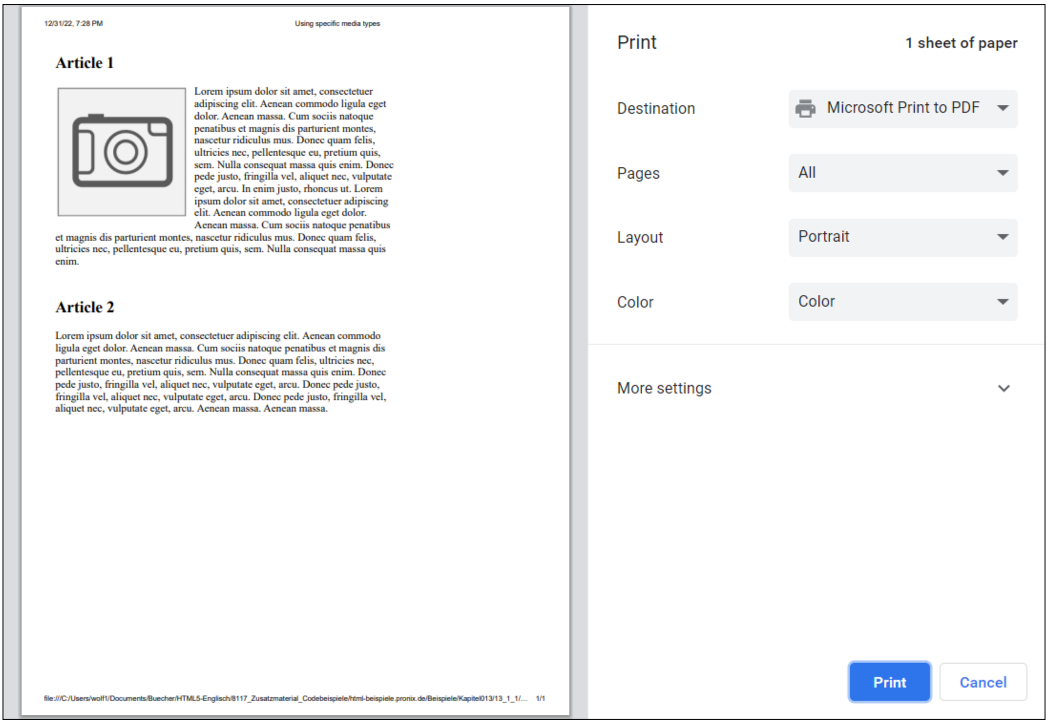
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Footer

Figure 13.1 The Web Page Was Styled with the CSS Version for the Screen (“media="screen"”)

The second version with *print.css*, also provided with the `link` element, is intended for the printer (`media="print"`) as the output medium. Without going into the content of the CSS file here, color has been omitted from this version, and borders have been removed. In addition, the `<header>`, `<nav>`, `<aside>`, and `<footer>` elements were hidden using `display: none` to print only the actual content of the `article` elements. You can see the version for the printer with *print.css* in Figure 13.2.

If, on the other hand, the media type isn’t defined, then the CSS statements automatically apply to all output types; this corresponds to `media="all"`.



12/31/22, 7:28 PM Using specific media types

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. Aenean massa. Aenean massa.

file:///C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/html-beispiele.pronix.de/Beispiele/Kapitel013/13_1_1/... 1/1

Print 1 sheet of paper

Destination: Microsoft Print to PDF

Pages: All

Layout: Portrait

Color: Color

Figure 13.2 The print.css Version for the Printer (“media="print"”) in Use

The two CSS files, *print.css* and *screen.css*, for this example can be found in */examples/chapter013/13_1_1/css/*.

Defining Media-Specific Sections via CSS Rule “@media”

Within a CSS file, you can use `@media [media type]` to define the CSS properties for different media types in curly brackets:

```
@media screen {
  /* CSS features for the screen */
}
@media print {
  /* CSS features for the printer */
}
```

You can also combine formatting in CSS with an `@media` rule within a `style` element in the HTML document:

```
...
    <style>
      @media screen {
        /* CSS features for the screen */
        ...
      }
      @media print {
        /* CSS features for the printer */
        ...
      }
    <style>
...
```

13.1.2 Media Queries for Media Features

In addition to media types, CSS can also be used to perform *media queries* of *media features*. Such queries are at the same time the heart of responsive web design. This allows you to make media queries—for example, regarding the size of the device, screen resolution, orientation (portrait or landscape), or input options (mouse, touch, keyboard, speech)—and respond accordingly with an appropriate design.

13.1.3 Integrating and Applying Media Queries for Media Features

The media features can be integrated and used in different ways. For example, you can write the use of such a media query in HTML as follows:

```
...
<head>
  <link rel="stylesheet" href="css/basis.css">
  <link rel="stylesheet" media="screen and (max-width: 480px)"
        href="css/mobile.css">
</head>
...
```

Here, *mobile.css* is used only if the maximum screen width of 480 pixels doesn't get exceeded. For devices with a higher resolution, only *basis.css* will be used. Older web browsers that use this media query (here, `media="screen and (max-width: 480px)"`), ignore this query and always use *basis.css*—even if the screen is less than 480 pixels wide.

It's also important to note that when using media queries, such as the one with the `link` element, all existing stylesheets will be downloaded, even if they don't apply to the query at all, but they won't be executed. So, in the preceding example, *base.css* and *mobile.css* are always loaded. The reason for this is to prevent a possible delay due to reloading when the web browser window gets resized or the orientation of the smartphone or tablet changes.

Integrating queries in the opening `<style>` tag is possible as follows:

```
...
<style type="text/css" media="screen and (max-width: 480px)">
   /* CSS statements for screen up to max. 480 pixels */
<style>
...
```

Besides the `link` and `style` elements, you can also write media queries of features as `@media` rules within a stylesheet:

```
...
.mainarticle {
  background-color: yellow;
}
@media screen and (max-width: 480px) {
  .mainarticle {
    background-color: orange;
  }
}
...
```

Here, the `@media` rule to color the background of `.mainarticle` orange is used only if the maximum screen size hasn't exceeded 480 pixels. Otherwise, the background of `.mainarticle` will be colored yellow.

Finally, the media feature queries can be used with the `@import` rule as follows:

```
@import url('css/mobile_480.css') screen and (max-width: 480px);
```

Thus, you can use media feature queries in *HTML* with the `link` element or in the `style` element, and in *CSS* with the `@media` or `@import` rule.

13.1.4 Basic Structure of a Media Feature Query

Now that you know how to use media queries in HTML or in CSS, let's take a closer look at the structure of such a query. For this purpose, we want to decompose the query `screen and (max-width: 480px)` into its individual components.

In Figure 13.3, you can see a media query and its individual components. Such a query consists of a **Media Type** (or output device) followed by a **Link** with `and`. Inside the **Expression**, a **Media Feature** (or property) and a corresponding **Value** are written between parentheses. I already described and listed the media types or the output devices in Chapter 8, Section 8.3.8.

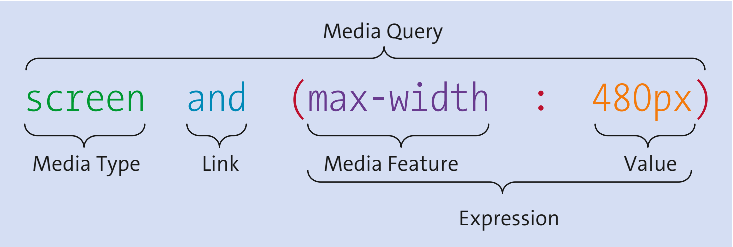


Figure 13.3 Individual Components of a Media Query

Linking the Media Features

The media feature gets linked via the keyword `and`. It's quite possible to link and process several `and` features. Linking can be done with and without a media type. Accordingly, a theoretical example of multiple links may look like the following:

```
@media screen and (min-width: 960px) {
  /* CSS statements for desktop */
}
@media screen and (min-width: 768px) and (max-width: 960px) {
  /* CSS statements for tablets and netbooks */
}
@media screen and (max-width: 480px) {
  /* CSS statements for smartphones */
}
```

In the second media query, the stylesheet in between is used only if all expressions and criteria linked via `and` are satisfied. In the example, the media type must be a screen, *and* the screen width must be at least 768 pixels *and* no more than 960 pixels.

If you use a media type, you can add a specification with `only` in front of the media type. With `only`, you make sure that older web browsers can't do anything with the media query. Sounds quite pointless, but it isn't. First, here's an example without `only`:

```
...
@media screen and (max-width: 480px) {
  /* CSS statements for smartphones */
}
...
```

A very old web browser may not be able to do anything with `and (max-width: 480px)` but is familiar with `@media screen`. To make sure that the web browser will ignore the specification of `and (max-width: 480px)` so that the CSS statements for smartphones will also be used on a desktop, you can put the keyword `only` in front of `screen` because older web browsers then won't know what to do with the query:

```
...
@media only screen and (max-width: 480px) {
  /* CSS statements for smartphones */
}
...
```

If you've used a media type, you can also prefix `not` and thus negate a query.

13.1.5 Which Media Features Can Be Queried?

The different output devices have many different features. Without a doubt, the most frequently used feature that's queried is the minimum and maximum width of the display area. Table 13.1 contains an overview of the most important media features that can be queried. An overview of all media features can be found at *www.w3.org/TR/mediaqueries-4/*.

Media Feature	Meaning	Values

`width`
`min-width`
`max-width`
	Width of the display area (viewport) of the web browser. Possible values are positive length values. Example: `(min-width: 480px)`	px, %, em

`height`
`min-height`
`max-height`
	Height of the display area (viewport) of the web browser. Possible values are positive length values. Example: `(max-height: 720px)`	px, %, em

`orientation`	This enables you to query the orientation of the device. The orientation can be `portrait` or `landscape`. In portrait mode, the value of `height` is higher than that of `width`. In landscape format, it's the other way around. Example: `(orientation: landscape)`	`portrait`, `landscape`

`aspect-ratio`
`min-aspect-ratio`
`max-aspect-ratio`
	Specifies the aspect ratio of `width` and `height` to each other. A value of 1,280 × 720 corresponds to an aspect ratio of 16:9, which you can address with `(aspect-ratio: 16/9)`. This corresponds to the specification, `(aspect-ratio: 1280/720)`.	Width/height, for example, `16/9`, `1280/720`

Table 13.1 Some Common Media Features That Can Be Queried via Media Queries

Media Feature	Meaning	Values
`color` `min-color/max-color`	Query for the color depth of the device. For black-and-white devices, the value is `color:0`.	Integer value (integer)
`color-index`, `min-color-index`, `max-color-index`	Checks the use of indexed colors of the output device.	Integer value (integer)
`monochrome`, `min-monochrome`, `max-monochrome`	Checks if the output device is monochrome. The value `monochrome:0` wouldn't be a monochrome device.	Integer value (integer)

`resolution`
`min-resolution`
`max-resolution`
	Query for the pixel density of the device, for example, `(resolution: 72dpi)`	`dpi`, `dcm`

`pointer`, `any-pointer`	Tests if the output device provides a mouse as an input device (or any input device at all).	`none` (device has only a keyboard) `coarse` (device has an input device with limited precision such as touch) `fine` (device has an input device with high accuracy such as mouse, touchpad)
`hover`, `any-hover`	Checks if the output device provides hover effects with the primary input device.	`none`, `hover`

Table 13.1 Some Common Media Features That Can Be Queried via Media Queries (Cont.)

The "min" and "max" Prefixes

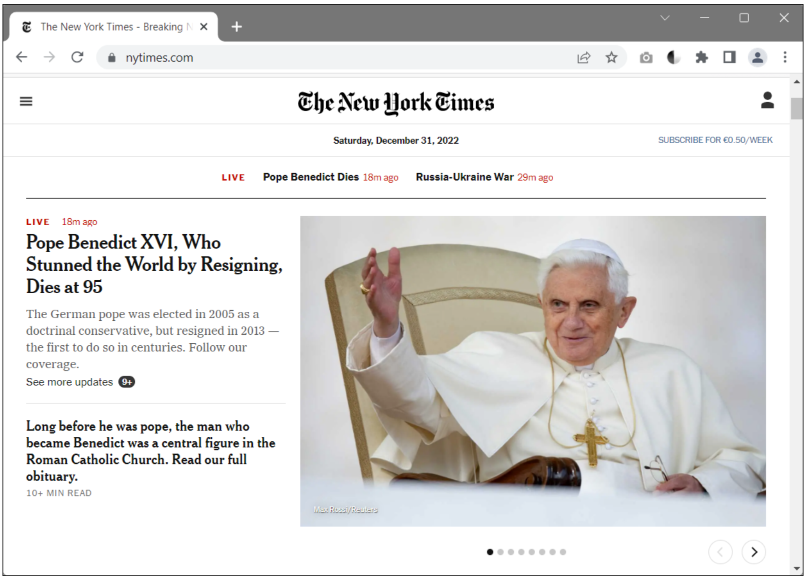
Particularly for the media features for the display area, it's usually more useful to use the versions with the `min` or `max` prefixes because one rarely knows the exact width of the user's display. For example, instead of using a media query where the exact width is queried with `width`, you should prefer the `min-width` and/or `max-width` version, which reacts even if the display width is at least or at most equal to the passed value.

13.1.6 Crucially Important: The Viewport for Mobile Devices

Especially in terms of querying media features from mobile devices, the viewport plays an essential role in ensuring that responsive web design works as intended there. Here, the viewport on desktop computers and the viewport on mobile devices often cause some confusion. The fact that high-resolution displays have an increasing market share makes things even more complicated because a pixel is suddenly no longer a pixel. A look at the website *http://screensiz.es/* will show you a collection of the many different sizes of displays on different devices. In the collection, you'll also find different width specifications for "`width`" and "`device-width`", which is due to the high-resolution displays just mentioned. I'll keep it as simple as possible at this point and won't bother you with the various viewport terms. Rather, I'll show you how to solve the problem with a single line.

In terms of desktop computers, the viewport is the inner area of the browser window without the borders. When you reduce or enlarge the browser window, the viewport gets reduced or enlarged too. You can address this visual viewport with the media features, `width` and `height`. On mobile devices such as a smartphone, the screens are much smaller than on a desktop computer, but the viewport there is often larger than on desktop screens. Therefore, without special adaptations of the viewport for mobile devices, the website would be displayed on these devices in the width of a typical desktop screen. The viewport on mobile devices is often referred to as the *layout viewport*.

In Figure 13.4, you can see the website *www.nytimes.com* on a desktop screen, and, in Figure 13.5, you see the same website in the Safari browser of an iPhone where the viewport hasn't been adapted for mobile devices. On the mobile device, the view has therefore been automatically scaled to fit the entire screen, just like on a desktop screen. Here, the layout viewport of the mobile device is scaled to the visual viewport, so to speak. As a result, you get unreadable text and everything in a thumbnail view that's much too small. To read the text and reports on the mobile device, you would need to zoom in and then scroll up and down or right and left.

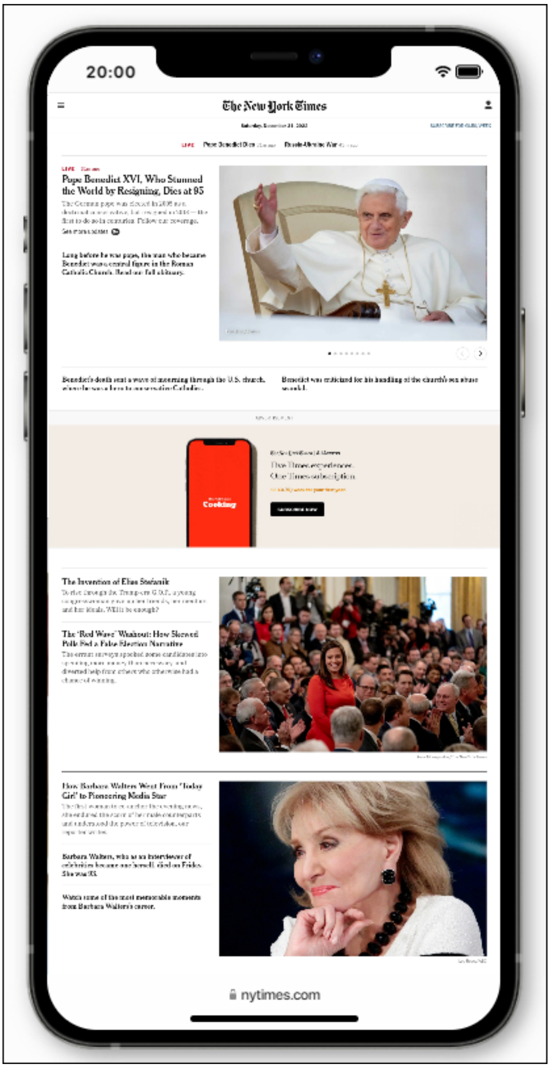


Pope Benedict XVI, Who Stunned the World by Resigning, Dies at 95

The German pope was elected in 2005 as a doctrinal conservative, but resigned in 2013 — the first to do so in centuries. Follow our coverage.

Long before he was pope, the man who became Benedict was a central figure in the Roman Catholic Church. Read our full obituary.

Figure 13.4 The New York Times Website on an Ordinary Desktop Screen



Pope Benedict XVI, Who Stunned the World by Resigning, Dies at 95

Long before he was pope, the man who became Benedict was a central figure in the Roman Catholic Church. Read our full obituary.

The Invention of Elise Stefanik

The 'Red Wave' Washout: How Skewed Polls Fed a False Election Narrative

How Barbara Walters Went From 'Today Girl' to Pioneering Media Star

Watch some of the most memorable moments from Barbara Walters's career.

Figure 13.5 The New York Times Website without a Customized Viewport on a Smartphone

The mobile issue can be easily fixed with the `viewport` metatag or the CSS rule, `@viewport`. To do this, you need to add the following to the `head` section of the HTML document:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

You can use `width=device-width` to set the width of the layout viewport to the width of the visual viewport. This line normalizes all different layout viewports from different devices and adjusts them to the current display size. You then don't have to worry about the different display sizes and can devote yourself to responsive web design in peace. In addition to `width`, there's the counterpart `height=device-height` for the height, which you'll rarely need in practice. For the viewport setting `width`, you can also use pixel values.

Another important viewport feature is `initial-scale:1.0`, which lets you set the initial zoom value to 100% or 1:1. Here, you can also define smaller or larger values as the initial zoom level.

In addition to `initial-scale`, there are further viewport features such as `minimum-scale` or `maximum-scale`, which enable you to define the minimum and maximum zoom level. `user-scalable=no` even allows you to disable zooming completely. This may be convenient for web apps, but for websites, zoom-level restrictions aren't recommended. You should keep in mind that there are website visitors who are depending on a high zoom level, and you would exclude those people from the website.

"@viewport" Rule

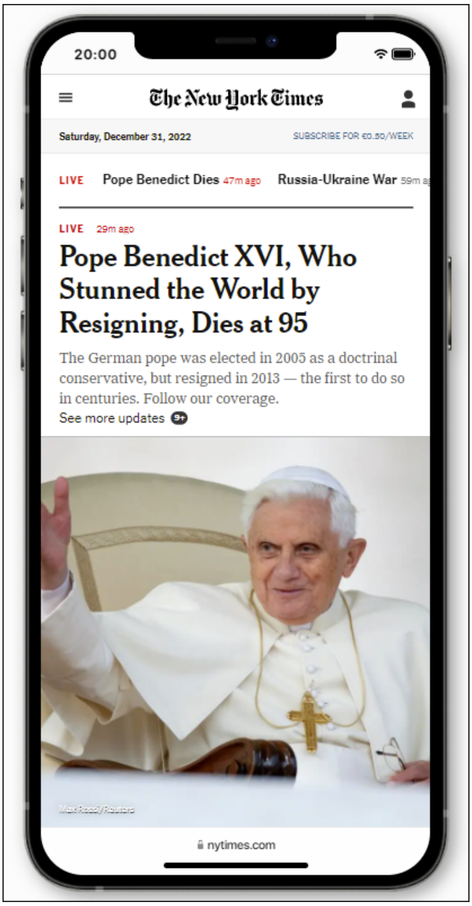
Going forward, the CSS rule `@viewport` will probably replace the metatag. Currently, however, the browser support for it is still very low (*https://caniuse.com/#search=%40viewport*). The advantage of the CSS rule is that theoretically more options can be used with it than with the viewport metatag, and thus it can be declared with different specifications in different media queries. The following is the equivalent counterpart to the preceding viewport metatag:

```
@viewport {
  width:device-width;
  zoom:1;
}
```

This specification corresponds to the following:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Of course, if you've customized the viewport on mobile devices, you still need to provide a rendering of the website that has been optimized for media queries. The viewport setting alone simply ensures that you turn off the web browser's automatic scaling and take everything into your own hands from now on. You'll learn how to create the responsive layout for it in the following sections. With regard to the website *www.nytimes.com* and the descriptions shown earlier in Figure 13.5, the website looks as shown in Figure 13.6 with the viewport adapted for mobile devices. Of course, the web developers of *www.nytimes.com* also provided an extra version for mobile devices via media queries, which I disabled in Figure 13.5 for demonstration purposes.



Saturday, December 31, 2022

SUBSCRIBE FOR €0.50/WEEK

LIVE Pope Benedict Dies 47m ago Russia-Ukraine War 59m a

LIVE 29m ago

Pope Benedict XVI, Who Stunned the World by Resigning, Dies at 95

The German pope was elected in 2005 as a doctrinal conservative, but resigned in 2013 — the first to do so in centuries. Follow our coverage.

See more updates 9+

nytimes.com

Figure 13.6 The New York Times Website with Adapted Viewport for Mobile Devices

13.1.7 Use "em" Instead of Pixels for a Layout Break in Media Queries

What's also quite useful is to perform the media queries with the `em` unit. This may seem weird at first glance because the screen is actually measured in pixels. But the advantage here is that the media query then works correctly even if the font size gets changed via the operating system or the **Zoom Text Only** function in Firefox. This ensures that when the fonts are displayed in a larger text size, the next layout level will actually be triggered, and the layout won't collapse. For example, for the website in Figure 13.7, the text was zoomed to the maximum using the **Zoom Text Only** feature in Firefox. In Figure 13.8, you can see what happens when pixels are used here instead of `em`. In Figure 13.9, however, `em` was used as the unit for layout wrapping in the media queries, and now the layout also reacts to the enlarged text display and selects the next layout level. Again, of course, this is assuming that an appropriate layout has been provided for it.

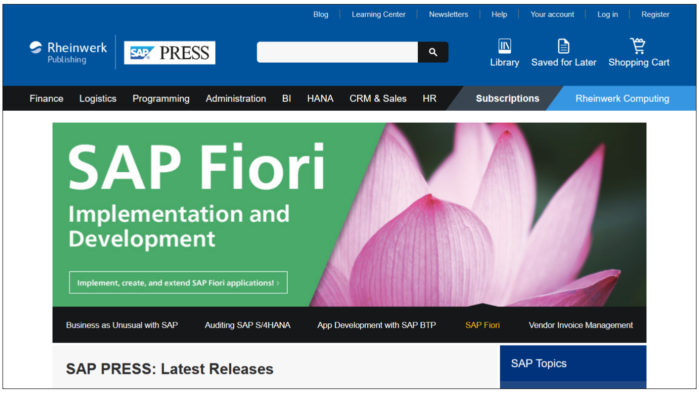

Blog
Learning Center
Newsletters
Help
Your account
Log in
Register
Rheinwerk Publishing
SAP PRESS
Library
Saved for Later
Shopping Cart
Finance
Logistics
Programming
Administration
BI
HANA
CRM & Sales
HR
Subscriptions
Rheinwerk Computing
SAP Fiori
Implementation and Development
Implement, create, and extend SAP Fiori applications! ›
Business as Unusual with SAP
Auditing SAP S/4HANA
App Development with SAP BTP
SAP Fiori
Vendor Invoice Management
SAP PRESS: Latest Releases
SAP Topics

Figure 13.7 The Website after Loading in the Firefox Web Browser

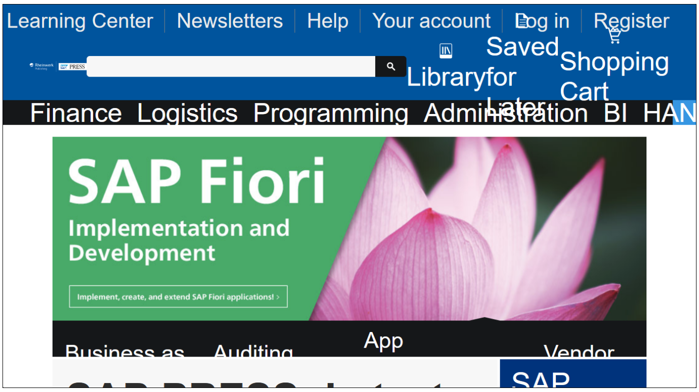

Learning Center
Newsletters
Help
Your account
Log in
Register
Library
Saved for Later
Shopping Cart
Finance
Logistics
Programming
Administration
BI
HAN
SAP Fiori
Implementation and Development
Implement, create, and extend SAP Fiori applications! ›
Business as
Auditing
App
Vendor
SAP

Figure 13.8 Here, the "Zoom Text Only" Function Was Used, but Pixels Were Used for the Layout Wrap in the Media Queries: The Layout Is Gone

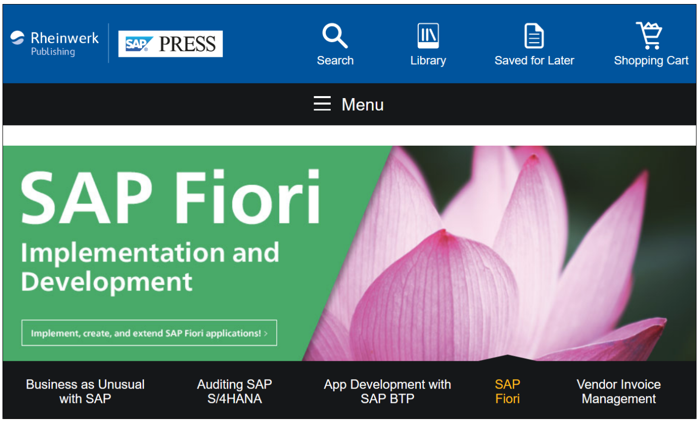

Rheinwerk Publishing
SAP PRESS
Search
Library
Saved for Later
Shopping Cart
Menu
SAP Fiori
Implementation and Development
Implement, create, and extend SAP Fiori applications! ›
Business as Unusual with SAP
Auditing SAP S/4HANA
App Development with SAP BTP
SAP Fiori
Vendor Invoice Management

Figure 13.9 This Is What It Should Look Like When the "Zoom Text Only" Function Is Executed and the "em" Unit Is Used in the Layout Break of the Media Queries: The Mobile Layout Is Now Executed Here

Consider the following media query:

```
@media screen and (min-width: 640px) {
   /* CSS statements for screens 640 pixels wide and larger */
}
```

Here, you've set up a layout break (also called *breakpoint*) for screens of 640 pixels or more. All CSS statements between the curly brackets are thus only executed from a screen width of 640 pixels. With reference to the recommendation to use the `em` unit for such layout breaks, you only need to divide the screen width by 16. The default browser base font size is usually 16 (pixels) and is therefore used as a reference size. As 640 pixels divided by 16 pixels/`em` equals 40 `em`, you can use the following `em` specification for the layout break of 640 pixels:

```
/* 640px / 16px/em = 40em */
@media screen and (min-width: 40em) {
   /* CSS statements for screens 640 pixels wide and larger */
}
```

13.1.8 Layout Breaks (Breakpoints)

In the previous section, I briefly mentioned a layout breakpoint. Such layout breaks are essential for the flexibility of a website. With these breaks, the layout gets changed. In practice, you provide different layouts for different resolutions, which you can control by means of media queries. Several such layout breaks using media queries can be defined, for example, as follows:

```
/* CSS statements for screens up to 640 pixels wide */

/* 640px / 16px/em = 40em */
@media screen and (min-width: 40em) {
   /* CSS statements for screens 640 pixels wide and larger */
}
/* 1024px / 16px/em = 64em */
@media screen and (min-width: 64em) {
   /* CSS statements for screens 1024 pixels wide and larger */
}
/* 1280px / 16px/em = 80em */
@media screen and (min-width: 80em) {
   /* CSS statements for screens 1280 pixels wide and larger */
}
```

In this example, I've defined three common layout breaks using media queries. The statements before the first layout break will be executed in any case. Here, in addition to the basic CSS features, you could also define the mobile layout for the smartphones right away. Then you'll find customized layouts for screen widths of 640 pixels (tablets), 1,024 pixels (desktop), and 1,280 pixels (extra-large desktop) as recommended with the `em` unit. According to the window width, the instructions are executed between the curly brackets.

However, the example isn't intended to give the impression that you need to define so many layout breaks. Thus, it's quite common to define only one layout for a mobile version and another layout for all other screens.

13.1.9 No More Math Games Thanks to "box-sizing: border-box;"

To avoid the embarrassment of having to recalculate later when creating the layout, you should use the newer box model with `border-box` right away. This means you don't need to calculate `width`, `padding`, and `border`, as I demonstrated in Chapter 11, Section 11.1.6, and include this information at the same time. Especially with responsive web design, this is a considerable relief. I've already described the new box model in detail in Section 11.2.

For this reason, it's recommended to set the following CSS statements right at the beginning:

```
html {
  box-sizing: border-box;
}
*, *::before, *::after {
  box-sizing: inherit;
}
```

13.1.10 What Happens to Web Browsers That Don't Understand Media Queries?

Media queries are now understood by all mainstream web browsers. If you still encounter old web browsers or other clients that aren't able to process media queries, then those web browsers will use the base version of your website that you defined with a media query before the first layout break. For example, if you've created a mobile version as the base version, the web browsers that can't process media queries will get this mobile version. For this reason, it's always recommended to create a basic version with a media query before the first layout breakpoint.

13.2 Let's Create a Simple Responsive Layout

Now that you know the necessary basics about media queries and how to use them to query media features, it's time to create a small responsive layout. I won't go into all the details here, which are also important in responsive web design. This is purely about arranging the content of a website for specific screen sizes. For demonstration purposes, I'll first use the classic method with `float`. In Section 13.4, I'll show you the modern way using the CSS grid.

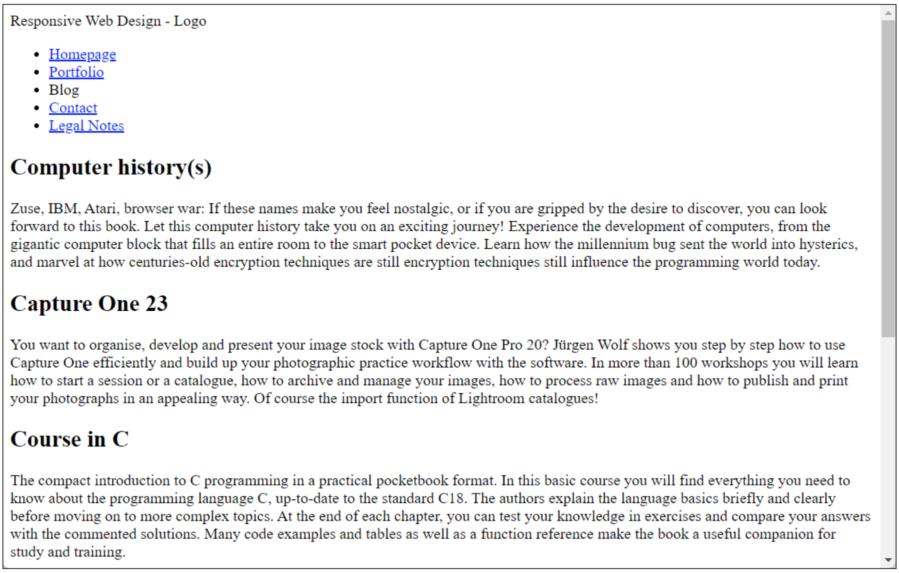
13.2.1 Let's Create the Basic Framework Using HTML

Before you can even begin to create the responsive layout with CSS, you must first write the basic framework with HTML. For this purpose, you can find a simple example in */examples/chapter013/13_2_1/index.html*, which contains basic HTML elements such as `<header>`, `<nav>`, `<main>`, `<aside>`, and `<footer>`. These five basic elements are then placed accordingly in the responsive layout. First, the basic HTML framework is as follows:

```
...
   <body>
      <header> Responsive Web Design—Logo </header>
      <nav>
         <ul>
            <li><a href="#">Homepage</a></li>
            <li><a href="#">Portfolio</a></li>
            <li>Blog</li>
            <li><a href="#">Contact</a></li>
            <li><a href="#">Legal Notes</a></li>
         </ul>
      </nav>
      <main>
         <article>
            <h2>LR Classic and PS</h2>
            <p>Get the most out of ... </p>
         </article>
         <article>
            <h2>Capture One 11</h2>
            <p>You want to increase your image stock ... </p>
         </article>
         <article>
            <h2>macOS High Sierra</h2>
            <p>In this comprehensive guidebook ... </p>
         </article>
         <article>
            <h2>PSE 2018</h2>
            <p>Whether photo optimization, image retouching ... </p>
         </article>
      </main>
      <aside>
         <h3>About the author</h3>
         <p>Jurgen Wolf is ... </p>
         <p> ... </p>
      </aside>
      <footer> Footer–&copy;&nbsp;2023 </footer>
   </body>
...
```

Listing 13.2 /examples/chapter013/13_2_1/index.html

We'll now add a responsive layout to the bare HTML framework in the following sections. You can see the basic framework without CSS in Figure 13.10 during execution.



Responsive Web Design - Logo

- Homepage
- Portfolio
- Blog
- Contact
- Legal Notes

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how centuries-old encryption techniques are still encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use Capture One efficiently and build up your photographic practice workflow with the software. In more than 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish and print your photographs in an appealing way. Of course the import function of Lightroom catalogues!

Course in C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the programming language C, up-to-date to the standard C18. The authors explain the language basics briefly and clearly before moving on to more complex topics. At the end of each chapter, you can test your knowledge in exercises and compare your answers with the commented solutions. Many code examples and tables as well as a function reference make the book a useful companion for study and training.

Figure 13.10 HTML Framework for Our Responsive Layout

13.2.2 Setting General CSS Features

Before you start worrying about layouts for specific screen widths, you should first write the general CSS features such as the box model, font size, color, and so on. These should be CSS features that don't change when a layout breakpoint exists. You can either write such general features in a separate CSS file that you add to the HTML document header, or write everything in a single CSS file, as I did in this example. As mentioned earlier, I'll keep this example very simple. Initially, I only activated the new box model via `box-sizing` and set the font in `body`.

Resetting/Normalizing CSS

It often happens that the CSS is reset/normalized at the beginning of the layout creation to bring the different basic browser settings of the various manufacturers to a common basis. This ensures that no differences exist between the different browsers. If you did a complete reset of the CSS features at the beginning of the web development, then usually the file *normalize.css* (from *https://necolas.github.io/normalize.css*) will be used to have a reasonable basis for the further use of CSS. Although I've omitted this in our example, it needed to be mentioned here.

13.2.3 What Should I Use as a Basic Version without Media Queries: Mobile First?

When you start creating the layout, you first need to think about what you want to start with. Do you want to start with the desktop version and then work your way down to the smaller versions for tablets and then smartphones via layout breaks and media queries? Or do you want to start with the mobile version for smartphones and then work your way up to the desktop version? This is entirely up to you and probably depends on the nature and content of your project.

Personally, I've come to prefer creating the mobile version first because (1) these devices are the most common medium that users use to get around the web, and (2) it forces you as a developer to stick to the basics first. And the essential thing on the web is the content. You could also say "Mobile first means content first." Furthermore, the desktop version also benefits from this because you can extend the uncomplicated display to this layout width as well, and the focus remains on the content.

Figure 13.11 shows the first draft of the mobile design planning. If you look at the basic HTML framework in */examples/chapter013/13_2_1/index.html* and Figure 13.10, you'll see that with this basic framework, you practically already have the mobile design. The only thing missing here is the styling with CSS.

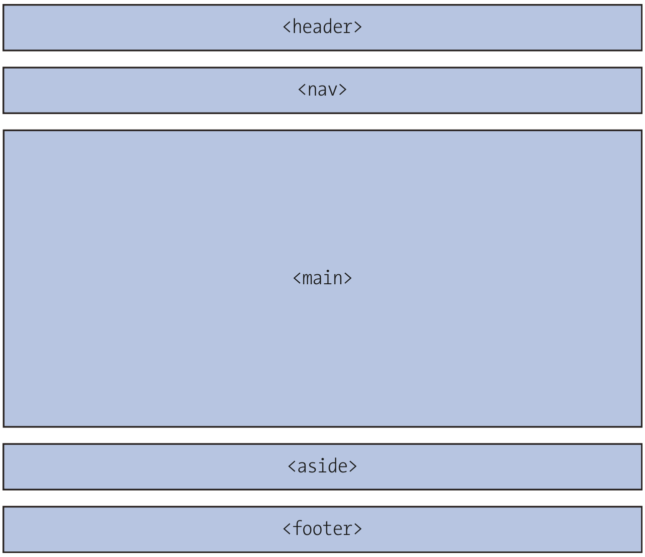


Figure 13.11 Design for the Mobile Version

The following CSS file (*/examples/chapter013/13_2_3/css/layout.css*) should therefore no longer hold any special surprises. Once you've set `box-sizing` with `border-box` to the new box model, you'll find the styling of each area for the mobile version.

```
@charset "UTF-8";
/* General basic settings */
html {
  box-sizing: border-box;
}
*, *::before, *::after {
  box-sizing: inherit;
}
body {
   color: #1d2731;             /* Ivory Black*/
   background-color: #efefef; /* Neutral */
   font-family: Georgia;
}
ul {
  padding: 0;
}
.wrapper {
   background-color: #ff3b3f; /* Watermelon */
}
.header {
   text-align: center;
   padding: 1em;
   background-color: #07889b; /* Teal */
   color: #efefef;            /* Neutral */
   border-bottom: 1px solid #efefef;
}
.aside {
  border-top: 1px solid #a9a9a9;
  padding-top: 0.5em;
}
.footer {
   background-color: #a9a9a9; /* Carbon */
   color: #efefef;            /* Neutral */
   padding: 1em;
   text-align: center;
   border-top: 1px solid #efefef;
}
.nav-ul {
   background-color: #ff383f; /* Watermelon */
   margin:0;
}
```

```
.nav-li {
   list-style: none;
   margin-left: 0;
   border-bottom: 1px solid #efefef;
}
.nav-li-a {
    padding: 0.6em 2rem;
    display: block;
}
.nav-ul a:link {
    text-decoration: none;
}
.nav-ul a:link, .nav-ul a:visited {
   color: #fff; /* white*/
}
.nav-ul a:hover, .nav-ul a:focus, .nav-ul a:active {
   background-color: #000; /* Black */
   color: #efefef; /* Neutral */
}
.nav-active {
   color: #000; /* Black */
   background-color: #fff; /* White */
}
.container {
   background-color: #fff; /* white */
   padding: 2em 2rem;
}
```

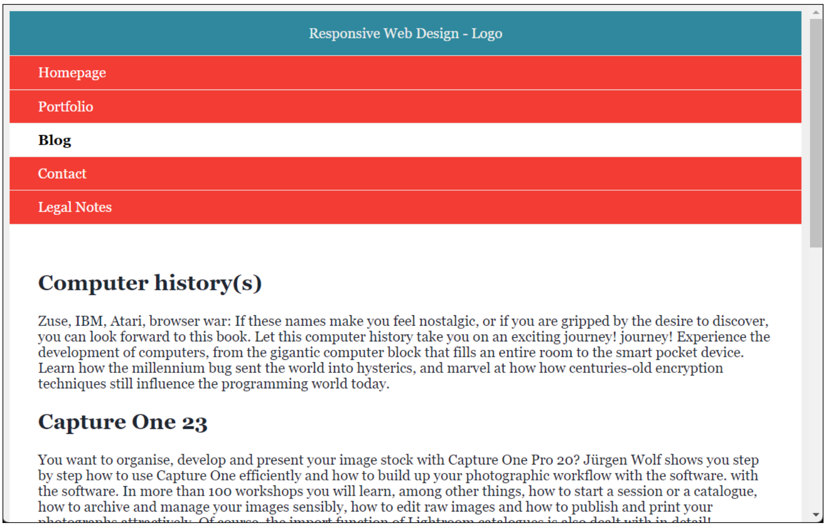
Listing 13.3 /examples/chapter013/13_2_3/css/layout.css

Next, you can enter the individual classes in the basic HTML framework. But before you do that, you should set the viewport metatag and add the CSS file for the layout. The basic HTML framework is thus already finished and won't get changed in the further course of this book. From now on, we'll work exclusively on the layout using the CSS file. Here's the finished basic HTML structure:

```
...
  <head>
    ...
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="css/layout.css">
  </head>
  <body>
    <div class="wrapper">
      <header class="header">Responsive Web Design—Logo</header>
      <nav>
        <ul class="nav-ul">
          <li class="nav-li"><a href="#" class="nav-li-a">Homepage</a></li>
          <li class="nav-li"><a href="#" class="nav-li-a">Portfolio</a></li>
          <li class="nav-li nav-active"><strong class="nav-li-a">Blog</strong></li>
          <li class="nav-li"><a href="#" class="nav-li-a">Contact</a></li>
          <li class="nav-li"><a href="#" class="nav-li-a">Legal Notes</a></li>
        </ul>
      </nav>
      <div class="container">
        <main class="content">
          <article class="article">
            <h2> Computer History </h2>
            <p>Zuse, IBM, Atari, browser war: ... </p>
          </article>
          <article class="article">
            <h2>Capture One 20</h2>
            <p>You want your ...</p>
          </article>
          <article class="article clear">
            <h2>Shell programming</h2>
            <p>Shell programming is ... </p>
          </article>
          <article class="article">
            <h2>Basic Course C</h2>
            <p>The compact introduction ... </p>
          </article>
        </main>
        <aside class="aside">
          <h3>About the author</h3>
          <p>Jurgen Wolf is ... </p>
          <p> ... </p>
        </aside>
      </div>
      <footer class="footer"> Footer - &copy;&nbsp;2023</footer>
    </div>
  </body>
</html>
```

Listing 13.4 /examples/chapter013/13_2_3/index.html

Without much effort, you've already created the layout for the mobile version. The layout now extends to 100% of the layout viewport on all devices. This mobile layout would also be used for (old) web browsers that can't do anything with media query layout breaks. At this point, you'll get the same layout on a tablet or desktop PC (see Figure 13.12) as on smartphone (see Figure 13.13). On the desktop computer, this single-column layout may not look spectacular, but it's clear, the content is rendered properly, and the website works.



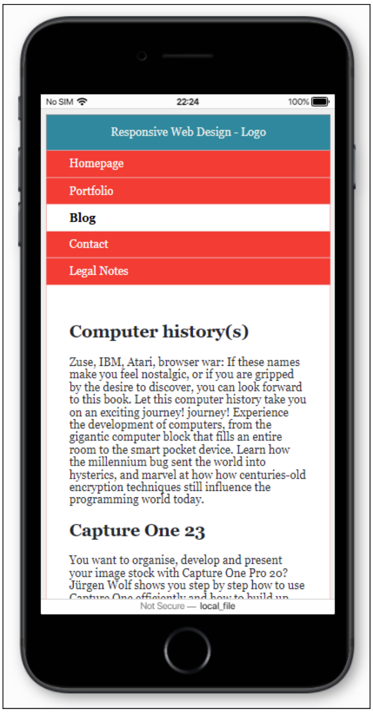
Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use Capture One efficiently and how to build up your photographic workflow with the software. with the software. In more than 100 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to edit raw images and how to publish and print your

Figure 13.12 The Basic Version without Media Queries on a Desktop Screen



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use

Figure 13.13 The Basic Version on a Smartphone, Which Is What It Was Created For

13.2.4 Setting the Layout Break (Breakpoint)

Even though the basic version provides visitors with all the basic functions of a website and the content gets displayed neatly, it's rather unusual to use this single-column layout for the tablet and desktop version as well. For this reason, we now want to provide another view of the layout, namely for tablets. With reference to our example, this means the layout should have two columns. The header, navigation, and footer can remain where they are. The main content and the sidebar, on the other hand, will now be positioned next to each other. Figure 13.14 shows the layout intended for a tablet.

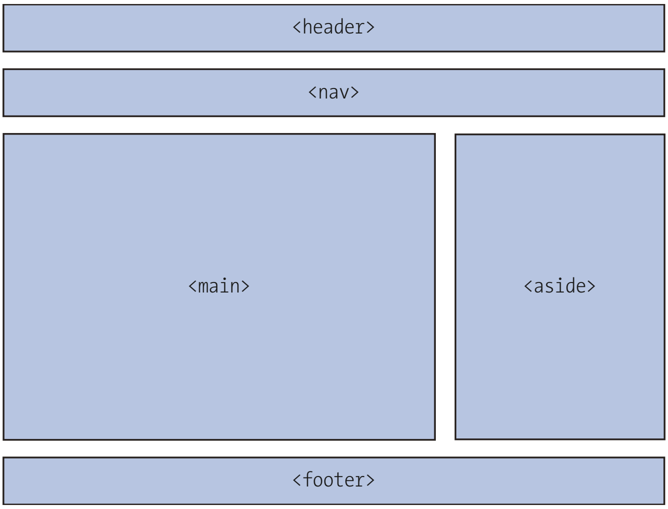


Figure 13.14 This Layout Is Intended for Tablets

First, you need to decide from which screen width onward you want to set the first layout break (breakpoint). In the following example, the breakpoint occurs at 640 pixels (= 40 em):

```
...
/* Tablet version from 640 pixels—640px / 16px/em = 40em */
@media screen and (min-width: 40em) {
  .header {
    padding: 1.5em;
    text-align: left;
  }
  .container {
    padding: 3rem 0;
    display: block;
    overflow: auto;
  }
  .content {
    display: block;
    float: left;
    width: 65%;
    padding: 0 1rem 0 2rem;
  }
  .aside {
    display: block;
    margin: 0 0 0 65%;
    width: 35%;
    padding: 0 2rem 0 2rem;
    border-top: none;
  }
  .nav-ul {
    padding: 0 2rem;
    overflow: hidden;
  }
  .nav-li {
      float: left;
      display: inline-block;
      border: none;
      width: auto;
  }
  .nav-li-a {
     padding: 0.7em 1.2rem;
     display: inline-block;
    }
}
```

Listing 13.5 /examples/chapter013/13_2_4/css/layout.css

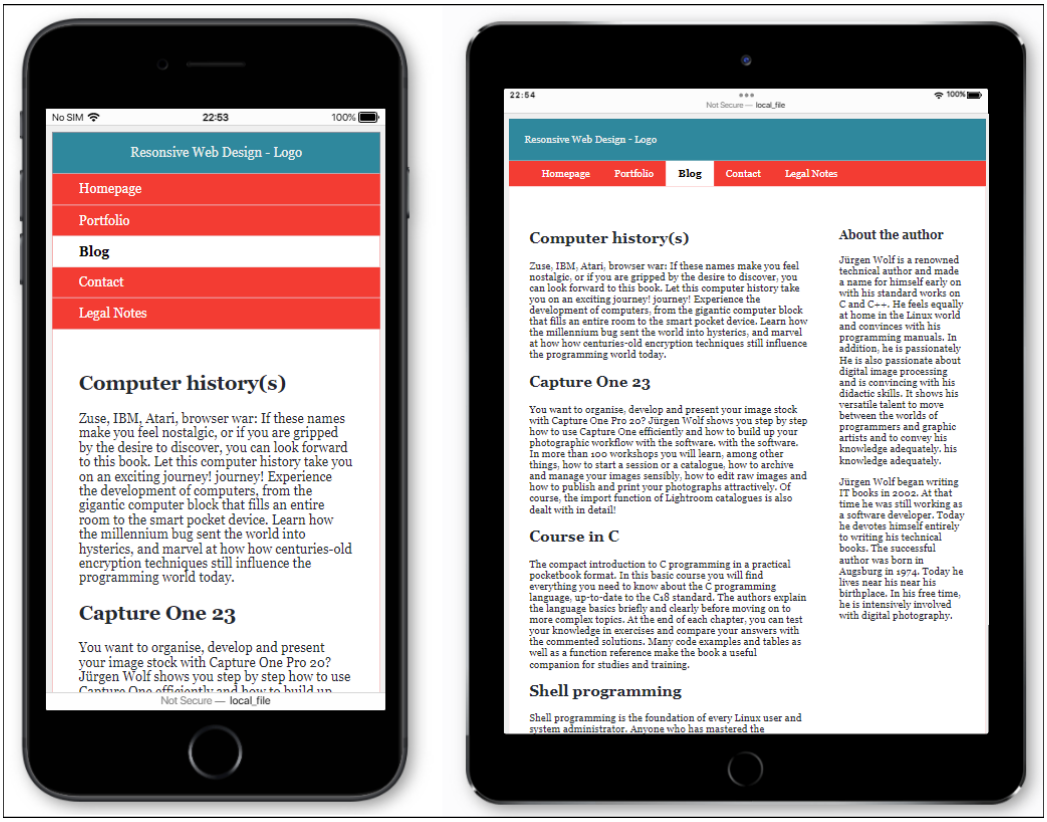
To get the main content and sidebar next to each other here, we provide the main content `.content` with a width of 65% and let the sidebar `.aside` flow next to it for the remaining 35% thanks to `float:left;` in `.content`. To prevent the other subsequent elements from "flowing along," we surround these two elements with a container (`.container`) and resolve the flowing around of the elements outside with `overflow:auto;` again. Here's the corresponding HTML part:

```
...
<div class="container">
  <main class="content">
    <article class="article">
      ...
    </article>
    ...
  </main>
  <aside class="aside">
```

```
    ...
  </aside>
</div>
...
```

Listing 13.6 /examples/chapter013/13_2_4/index.html

The layout break occurs from a viewport of 640 pixels onward. In Figure 13.15 you can still see the layout on a viewport with less than 640 pixels on the left, whereas on the right, the layout break has taken place because the screen width was more than 640 pixels. This layout break will also be executed on smartphones if you switch to landscape format and the width is more than 640 pixels in the process.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use Capture One efficiently and how to build up your photographic workflow with the software. with the software. In more than 100 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to edit raw images and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Course in C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the C programming language, up-to-date to the C18 standard. The authors explain the language basics briefly and clearly before moving on to more complex topics. At the end of each chapter, you can test your knowledge in exercises and compare your answers with the commented solutions. Many code examples and tables as well as a function reference make the book a useful companion for studies and training.

Shell programming

Shell programming is the foundation of every Linux user and system administrator. Anyone who has mastered the

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and convinces with his programming manuals. In addition, he is passionately He is also passionate about digital image processing and is convincing with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic artists and to convey his knowledge adequately. his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near his near his birthplace. In his free time, he is intensively involved with digital photography.

Figure 13.15 Now the Basic Version Is Switched to the Next Layout Version from a Screen Width of 640 Pixels

13.2.5 Adding More Layout Breaks

In our example, we could already be pretty happy with these two versions—a mobile version for screens smaller than 640 pixels and a second version for screens with a width of 640 pixels or more. However, we still need to add two more breakpoints for a desktop and an extra-wide desktop because on wider screens, the text lines of the main content would otherwise become much too long. To fill the space for a larger viewport, the navigation should now be added as a third column, so that it gets positioned to the left of the main content, while the sidebar is to the right of it. We'll leave the header and footer as they are. The design for desktop screen layout can be found in Figure 13.16.

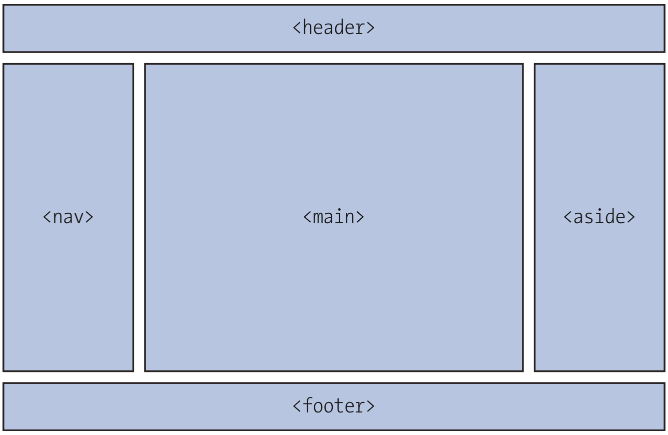


Figure 13.16 To Be Used for the Layout of Desktop Screens

We define the next layout break for screens with a wider viewport than 1,024 pixels (64 em). Here is the corresponding CSS code:

```
...
/* Screens from 1024 pixels—1024px / 16px/em = 64em */
@media screen and (min-width: 64em) {
  .container {
    width: 85%;
    padding: 0;
    margin-left: 15%;
  }
  .content {
    width: 70%;
    padding: 1em 1.5em;
  }
  .aside {
    width: 30%;
    padding: 1em 1.5em;
    margin: 0 0 0 70%;
  }
  .nav-ul {
    width: 15%;
    float: left;
    margin: 1em 0;
    padding:0;
  }
  .nav-li {
    width: 100%;
    float: none;
    text-align: center;
  }
  .nav-li-a {
    padding: 0.5em 3rem;
    display: block;
  }
}
```

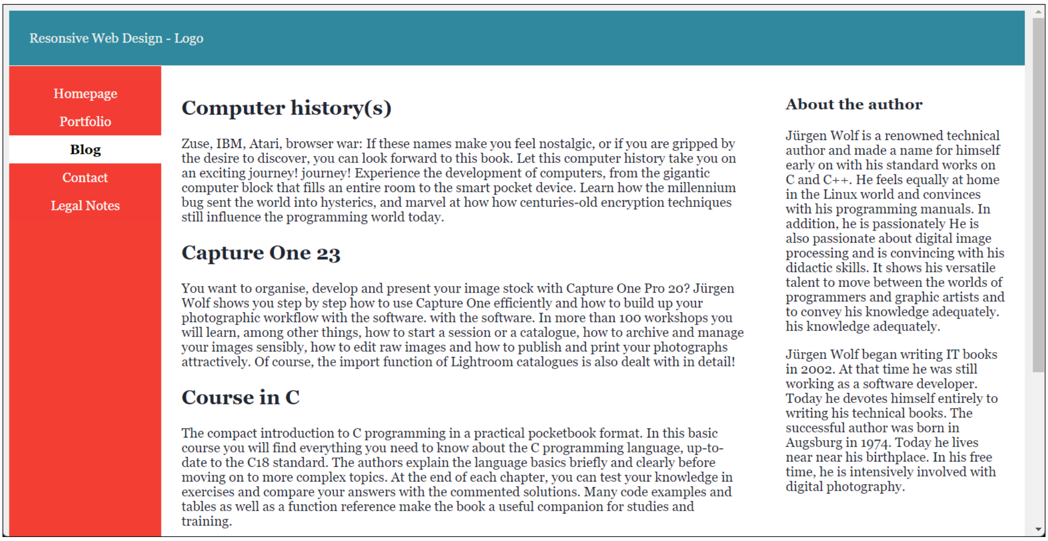
Listing 13.7 /examples/chapter013/13_2_5/css/layout.css

For the `.container` with the main content `.content` and the sidebar `.aside`, 85% of the window width has been reserved now. Of this 85% within `.container`, the main content (`.content`) shares this space with 70% and the sidebar `.aside` with 30%. The remaining 15% space is reserved for the navigation (`.nav`). For this purpose, space was left free for the container `.container` with `margin-left`. Due to `float:left;` at `.nav`, the container `.container` with its main content and sidebar is placed to the right of it. The HTML document with the corresponding passages is also shown in Listing 13.8 to illustrate what has just been described:

```
...
<nav>
  <ul class="nav-ul">
    <li class="nav-li"><a href="#" class="nav-li-a">Homepage</a></li>
    ...
  </ul>
</nav>
<div class="container">
  <main class="content">
    <article class="article">
      ...
    </article>
    ...
  </main>
  <aside class="aside">
    ...
  </aside>
</div>
...
```

Listing 13.8 /examples/chapter013/13_2_5/index.html

In Figure 13.17, you can see the layout for the desktop starting at a viewport width of 1,024 pixels during execution.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use Capture One efficiently and how to build up your photographic workflow with the software. with the software. In more than 100 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to edit raw images and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Course in C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the C programming language, up-to-date to the C18 standard. The authors explain the language basics briefly and clearly before moving on to more complex topics. At the end of each chapter, you can test your knowledge in exercises and compare your answers with the commented solutions. Many code examples and tables as well as a function reference make the book a useful companion for studies and training.

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and convinces with his programming manuals. In addition, he is passionately He is also passionate about digital image processing and is convincing with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic artists and to convey his knowledge adequately. his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near near his birthplace. In his free time, he is intensively involved with digital photography.

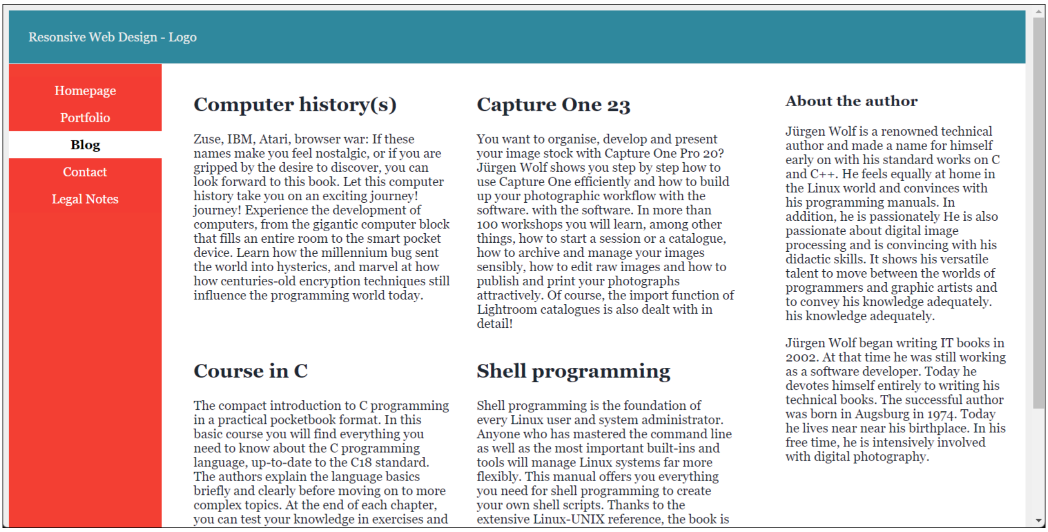
Figure 13.17 The Layout for the Desktop Version from a Viewport of 1,024 Pixels Wide

You should now define another breakpoint here for extra-large screens because, otherwise, the lines there will again become too long. Here, you can either decide that the complete layout must not extend further beyond a certain screen width, or you can split the articles of the main content. In the example, I've created a final breakpoint for a screen width of 1,280 pixels (80 em) or wider, which offers both solutions just mentioned. The `.wrapper` sets the `max-width` feature to 1,280 pixels when the width exceeds 1,280 pixels so that the website can't expand any further. Furthermore, I reduced the `.article` articles to 50% and arranged them next to each other via `float:left;`. However, you can also use the `.wrapper` or `.article` class alone for this example. Here's the corresponding CSS code for the last breakpoint:

```
...
/* Large screens (>1280 pixels)–1280px / 16px/em = 80em */
@media screen and (min-width: 80em) {
  .wrapper {
    margin: 0 auto;
    max-width: 80em;
  }
  .article {
    display: block;
    width: 50%;
    float:left;
    padding: 0 1rem 0 1rem;
  }
  .clear { clear: both; }
}
```

Listing 13.9 /examples/chapter013/13_2_5/css/layout.css

You can see the result with an extra-wide viewport from 1,280 pixels in Figure 13.18.



Resonsive Web Design - Logo

Homepage
Portfolio
Blog
Contact
Legal Notes

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use Capture One efficiently and how to build up your photographic workflow with the software. with the software. In more than 100 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to edit raw images and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Course in C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the C programming language, up-to-date to the C18 standard. The authors explain the language basics briefly and clearly before moving on to more complex topics. At the end of each chapter, you can test your knowledge in exercises and

Shell programming

Shell programming is the foundation of every Linux user and system administrator. Anyone who has mastered the command line as well as the most important built-ins and tools will manage Linux systems far more flexibly. This manual offers you everything you need for shell programming to create your own shell scripts. Thanks to the extensive Linux-UNIX reference, the book is

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and convinces with his programming manuals. In addition, he is passionately He is also passionate about digital image processing and is convincing with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic artists and to convey his knowledge adequately. his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near near his birthplace. In his free time, he is intensively involved with digital photography.

Figure 13.18 This Layout Is for extra-Large Screens of 1,280 Pixels or Wider

The example created in this way now flexibly adapts to visitors' devices thanks to the layout breaks with the media queries. You now have used an easy method to create a simple version for smartphones, a version for tablets, and another version for desktop PCs.

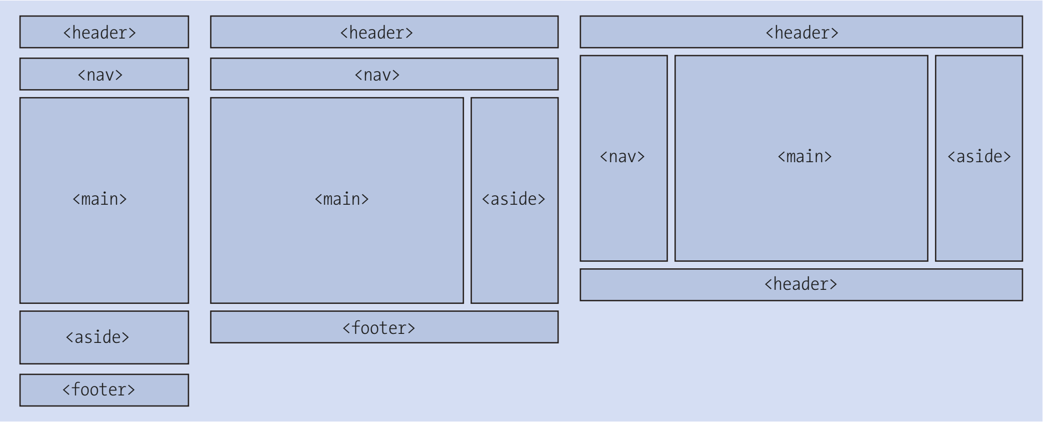
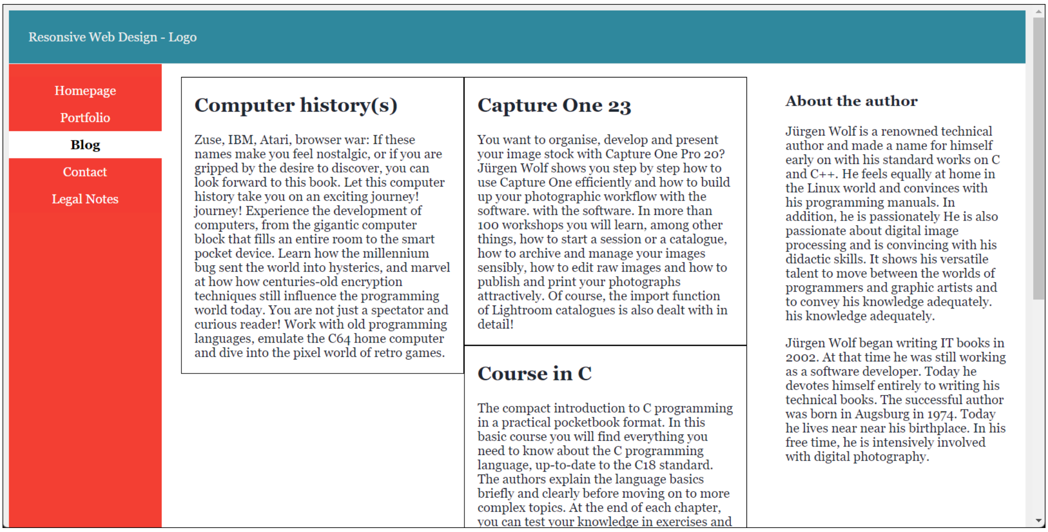


Figure 13.19 Multiple Layout Breaks for Different Screen Resolutions Thanks to Media Queries

13.2.6 Customizing the Main Content

Once you've created the responsive design, you can get to working on the details. In the example, we still want to keep an eye on the main content with the articles when the desktop width is extra large. If, for example, an article on the left-hand side is longer than usual, and you haven't taken any precautions, the article after the next one can't slide to the left and remains attached to the overlong article. Figure 13.20 shows such a case.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the millennium bug sent the world into hysterics, and marvel at how how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and dive into the pixel world of retro games.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to use Capture One efficiently and how to build up your photographic workflow with the software. with the software. In more than 100 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to edit raw images and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Course in C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the C programming language, up-to-date to the C18 standard. The authors explain the language basics briefly and clearly before moving on to more complex topics. At the end of each chapter, you can test your knowledge in exercises and

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and convinces with his programming manuals. In addition, he is passionately He is also passionate about digital image processing and is convincing with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic artists and to convey his knowledge adequately. his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near near his birthplace. In his free time, he is intensively involved with digital photography.

Figure 13.20 For Clarity, I've Highlighted the Articles with a Frame

In the */examples/chapter013/13_2_5/css/layout.css* example, I wrote a `.clear` class with `clear:both;` at the end for this. You only need to add this class to the corresponding article as follows:

```
...
<article class="article clear">
   ...
</article>
...
```

Listing 13.10 /examples/chapter013/13_2_5/index.html

This solves the problem shown in Figure 13.20, and the "sticking" article can "slide" back down, but this procedure is still quite tedious when you change the website and add new articles. Instead, it makes more sense to write a class for a row with articles that you can use with `<div class="row">` and that automatically includes a `clear:both;` at the end of the line. In practice, you could implement this as follows:

```
...
<main class="content">
  <div class="row">
    <article class="article">
       ...
    </article>
    <article class="article">
       ...
    </article>
  </div>
  <div class="row">
    <article class="article">
     ...
    </article>
    <article class="article">
     ...
    </article>
  </div>
</main>
...
```

Listing 13.11 /examples/chapter013/13_2_6/index.html

You've practically already defined the `.row` class for this at the layout break of 1,280 pixels with the designation `.clear`. It makes sense that it should be called `.row` here now. Here's the CSS code for it:

```
...
@media screen and (min-width: 80em) {
  .wrapper {
    margin: 0 auto;
    max-width: 80em;
  }
  .article {
    display: block;
    width: 50%;
    float:left;
    padding: 0 1rem 0 1rem;
  }
  .row { clear: both; }
}
```

Listing 13.12 /examples/chapter013/13_2_6/css/layout.css

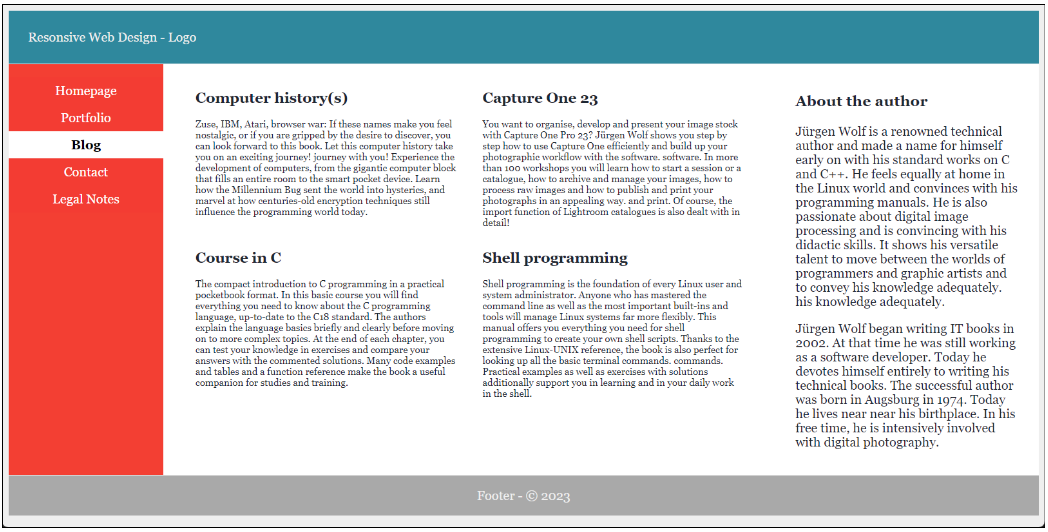
13.3 Even More Flexible Elements

With the fluid responsive layout you created on the previous pages, the development of a responsive website is far from being complete. The important topics that haven't been considered yet are typography, flexible image elements, and mobile navigation. This will be briefly described here, so that you at least know what's important and what else you should consider.

13.3.1 Use Relative Font Sizes instead of Pixels

Text design is a very important topic in web design and is therefore also covered separately in Chapter 14, Section 14.1. Nevertheless, I want to make a few remarks about it at this point. As with the media queries layout breaks, you should always use relative specifications instead of pixels for the font sizes. When zooming an entire page (page zoom), most web browsers have no problems if the specifications are made in pixels or relative units.

Even on screens with a higher pixel density, fonts that are specified with pixels are displayed relatively small. Although users could again zoom in afterwards, a website should be rendered legibly right after loading. For this reason, you should avoid font specifications in pixels and use relative specifications such as `em`, `rem`, or percent.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let this computer history take you on an exciting journey! journey with you! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Course in C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the C programming language, up-to-date to the C18 standard. The authors explain the language basics briefly and clearly before moving on to more complex topics. At the end of each chapter, you can test your knowledge in exercises and compare your answers with the commented solutions. Many code examples and tables and a function reference make the book a useful companion for studies and training.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 23? Jürgen Wolf shows you step by step how to use Capture One efficiently and build up your photographic workflow with the software. software. In more than 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish and print your photographs in an appealing way. and print. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Shell programming

Shell programming is the foundation of every Linux user and system administrator. Anyone who has mastered the command line as well as the most important built-ins and tools will manage Linux systems far more flexibly. This manual offers you everything you need for shell programming to create your own shell scripts. Thanks to the extensive Linux-UNIX reference, the book is also perfect for looking up all the basic terminal commands. commands. Practical examples as well as exercises with solutions additionally support you in learning and in your daily work in the shell.

About the author

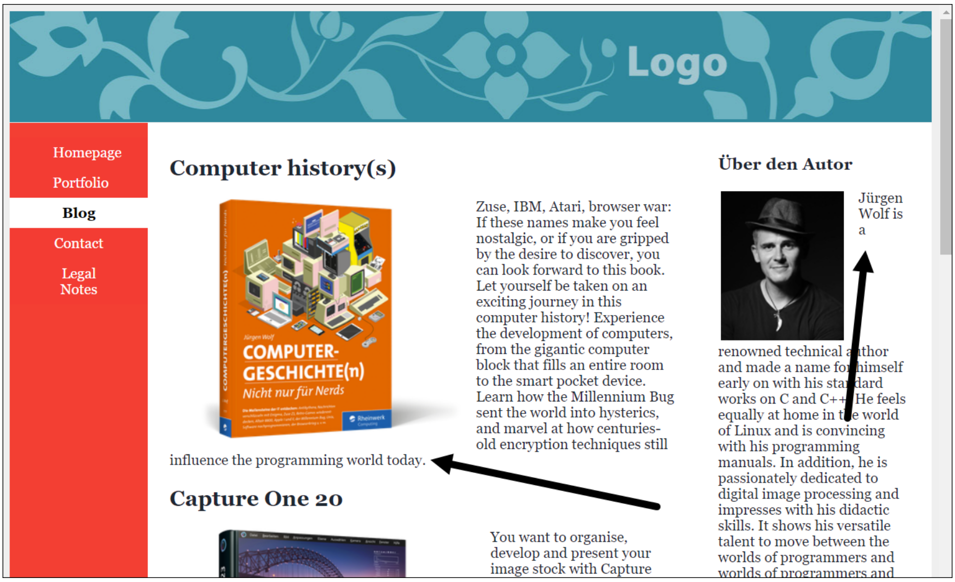
Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and convinces with his programming manuals. He is also passionate about digital image processing and is convincing with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic artists and to convey his knowledge adequately. his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near near his birthplace. In his free time, he is intensively involved with digital photography.

Figure 13.21 If the Font Size Is Wrong, the Best Responsive Layout Is Useless

13.3.2 Making Images Responsive

If you're going to add images to the responsive layout, you'll also want to make them responsive and not leave them rigid in width. In Figure 13.22, you can see what happens with rigid images when the screen becomes narrower.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 20

You want to organise, develop and present your image stock with Capture

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and

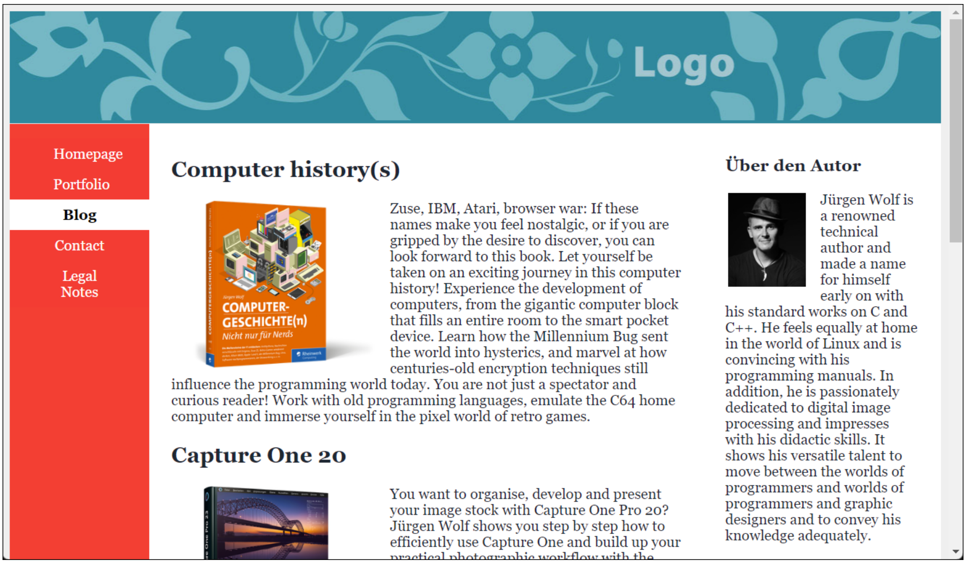
Figure 13.22 Flowing around the Text When the Image Size Is Rigid Can Cause the Text to Slip Away at the Bottom, and/or Individual Words Can Be Left at the Top If There Isn't Enough Space

Here, a few words may get stuck next to the image or slip away to the bottom, which is a normal behavior of `float`, but not very attractive. You can avoid this text wrapping by setting the maximum width of the image accordingly as a percentage using the CSS feature `max-width`, which allows you to define the maximum width of an element. In the example, the image should be allowed to absorb 40% in the `article` element. The same should apply to the author image in the `aside` element. For this purpose, I've extended the basic version with the class selectors `.img-art` and `.img-side`:

```
...
.img-art {
  float: left;
  margin: 0 1em 0.2em 0;
  max-width: 40%;
  height: auto;
}
.img-side {
  float: left;
  margin: 0.1em 1em 0.2em 0.2em;
  max-width: 40%;
  height: auto;
}
...
```

Listing 13.13 /examples/chapter013/13_3_2/css/layout.css

In Figure 13.23, the images in the article and sidebar now adjust by 40% to the article width and the sidebar, respectively. Because these selectors were written in the base version, these features apply to all layout widths. In Figure 13.24, you can see the layout for extra-wide desktops, and, in Figure 13.25, you can see the smartphone version. Of course, there's nothing that speaks against setting a separate width for the images for each layout break by using `max-width`.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

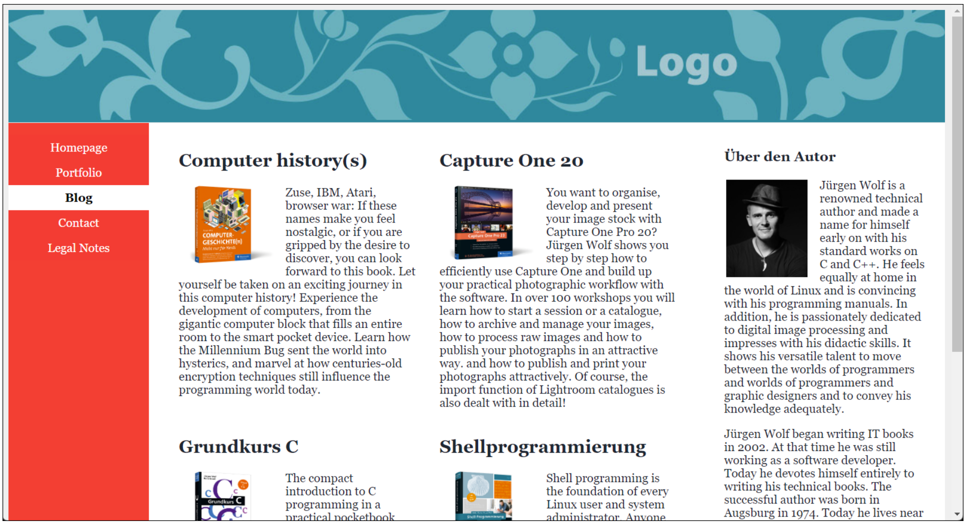
Capture One 20

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Figure 13.23 The Image Size Now Also Adapts to the Screen Width and Is Displayed Relative to the <article> or <aside> Element in the Corresponding Size (here, 40%)



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 20

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish your photographs in an attractive way. and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Grundkurs C

The compact introduction to C programming in a practical pocketbook

Shellprogrammierung

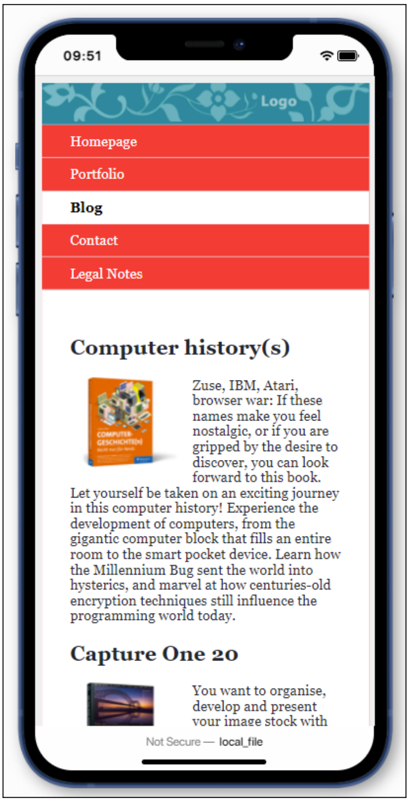
Shell programming is the foundation of every Linux user and system administrator. Anyone

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near

Figure 13.24 The 40% Image Width with an Extra-Wide Desktop



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 20

You want to organise, develop and present your image stock with

Figure 13.25 Responsive Images Also Pay Off on a Smartphone

In addition to using `max-width` for the width, you can use `height` to specify the height. Here, I've set `height` to `auto` so that the web browser automatically adjusts the height in proportion to the width.

You can simply add the classes in the HTML document as usual to the `<img>` tags of those images that should be responsive:

```
...
<div class="container">
  <main class="content">
    <article class="article">
      <h2>LR Classic and PS</h2>
      <p><img src="graphics/cover-3.png" alt="Cover" class="img-art">... </p>
    </article>
     ...
  </main>
  <aside class="aside">
    <h3>About the author</h3>
    <p><img src="graphics/me.jpg" alt="Cover" class="img-side"> ... </p>
  </aside>
</div>
...
```

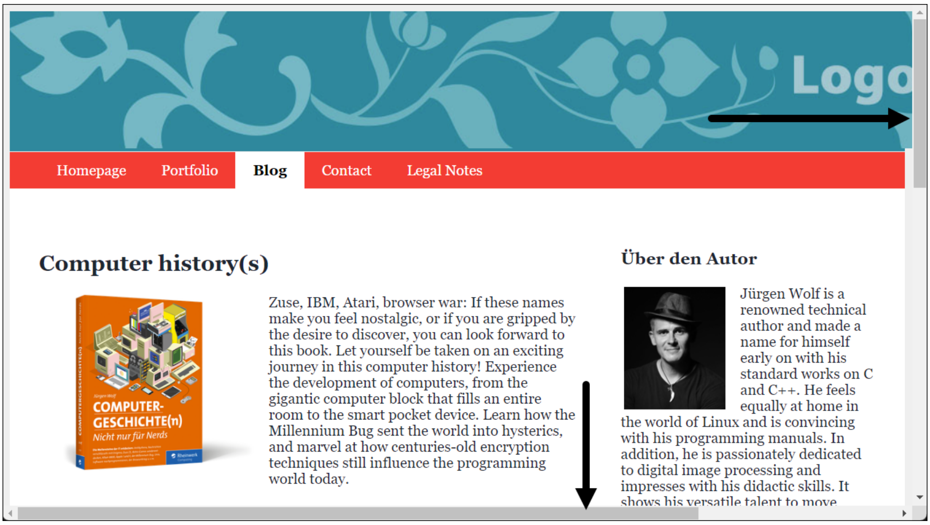
Listing 13.14 /examples/chapter013/13_3_2/index.html

13.3.3 Flexible Images in Maximum Possible Width

If you want images to always stretch across the full width regardless of the device and still leave them responsive, you can set `max-width` to 100%. In that case, it depends on where you place these images to make them responsive. As a matter of fact, setting `max-width` to 100% also means that an image won't respond until the column in which it has been defined is smaller than the image. This means that an image with a width of 300 pixels doesn't become responsive until the width where it's used is less than 300 pixels. So, setting an image with `max-width` to 100% depends on the context in which the image is used. Here's an example where you insert a graphic with a size of 1,280 × 150 pixels with `<img>` into the `header` element of the HTML document:

```
...
<header class="header">
  <img src="images/logo.jpg" alt="Logo">
</header>
...
```

Without any further preparations, the image would be displayed in full size; on screens smaller than 1,280 pixels, the logo would be cut off on the right, and a horizontal scroll bar would be displayed in the browser, as in Figure 13.26.



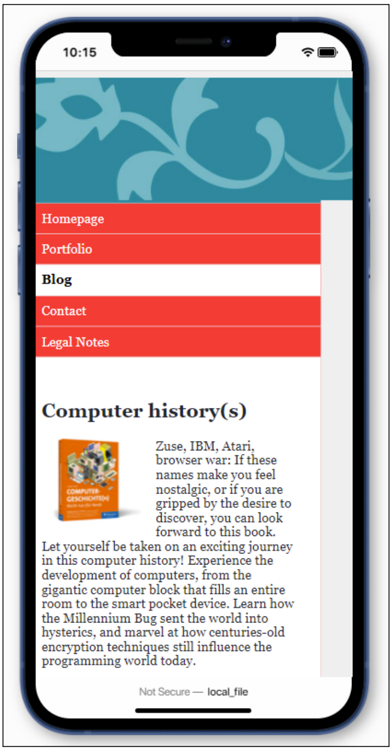
Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move

Figure 13.26 If the Browser Width Is Too Small, the Image Gets Cut Off and a Horizontal Scroll Bar Appears



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Figure 13.27 Things Don't Look Much Better in the Mobile Version

In this example, you can use CSS and `max-width: 100%` to respond as follows:

```
...
.img-logo {
  max-width: 100%;
  height: auto;
}
...
```

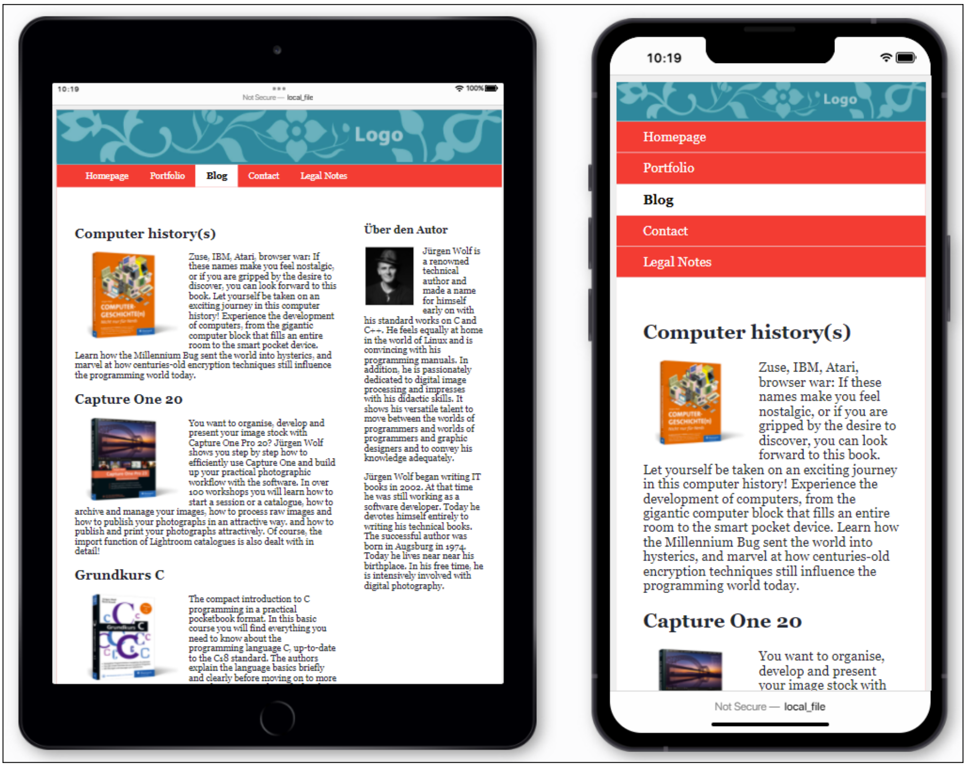
Listing 13.15 /examples/chapter013/13_3/css/layout.css

You merely need to add the class to the `img` element:

```
<header class="header">
  <img src="graphics/logo.jpg" alt="Logo" class="img-logo">
</header>
```

Listing 13.16 /examples/chapter013/13_3/index.html

In Figure 13.28, you can see how the image in the `<header>` responds to the appropriate screen width and is made flexible.



10:19

Not Secure — local_file

100%

Logo

Homepage Portfolio Blog Contact Legal Notes

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 20

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish your photographs in an attractive way. and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Grundkurs C

The compact introduction to C programming in a practical pocketbook format. In this basic course you will find everything you need to know about the programming language C, up-to-date to the C18 standard. The authors explain the language basics briefly and clearly before moving on to more

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near near his birthplace. In his free time, he is intensively involved with digital photography.

10:19

Logo

Homepage

Portfolio

Blog

Contact

Legal Notes

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 20

You want to organise, develop and present your image stock with

Not Secure — local_file

Figure 13.28 The Width for the Image in the <header> Adjusts for Tablets (Left) and the Width for the Image in the <header> Also Adjusts for Smartphones (Right)

Responsive Videos: <video>

Everything you can do with images, you can also do with videos that you've integrated via `<video>` using the same CSS instructions.

13.3.4 Hiding Images Entirely

If you look at Figure 13.28 with the logo in the header, it already looks relatively small and lost in the header. I would completely omit the logo in the header in this example for the smartphone version. To remove it from the smartphone version, that is, the basic version, you need to set `display` to `none`:

```
...
.img-logo {
  display: none;
}
...
```

Listing 13.17 /examples/chapter013/13_3_4/css/layout.css

Accordingly, for the other layout breaks, you should make the header visible again via `display:block` and set `max-width` to 100%:

```
...
.img-logo {
  display: block;
  max-width: 100%;
  height: auto;
}
...
```

Listing 13.18 /examples/chapter013/13_3_4/css/layout.css

13.3.5 Loading the Right Image for the Screen Width: <picture>

You now know how to make images responsive with `max-width`. The disadvantage of this method is that small displays often load files that are too large, which unnecessarily slows down performance and increases the amount of data transfer. Furthermore, on small displays, the images have to be scaled down again, which impacts the image quality. High-resolution displays aren't taken into consideration at all. If you've read the book from the beginning, you already know an alternative solution. In Chapter 6, Section 6.3, you already learned how to specify alternate image sources for different viewports using the HTML elements `<picture>` and `<source>` and media queries with the `media attribute`.

The `picture` element is an HTML element that serves as a container element for multiple image sources. You can specify the individual image sources using the `source` element. In the following example, we want to load an image source in the first article for the book cover that matches the display. To distinguish when which image gets loaded, I've added appropriate text to the graphics. High-resolution displays are taken into account right away.

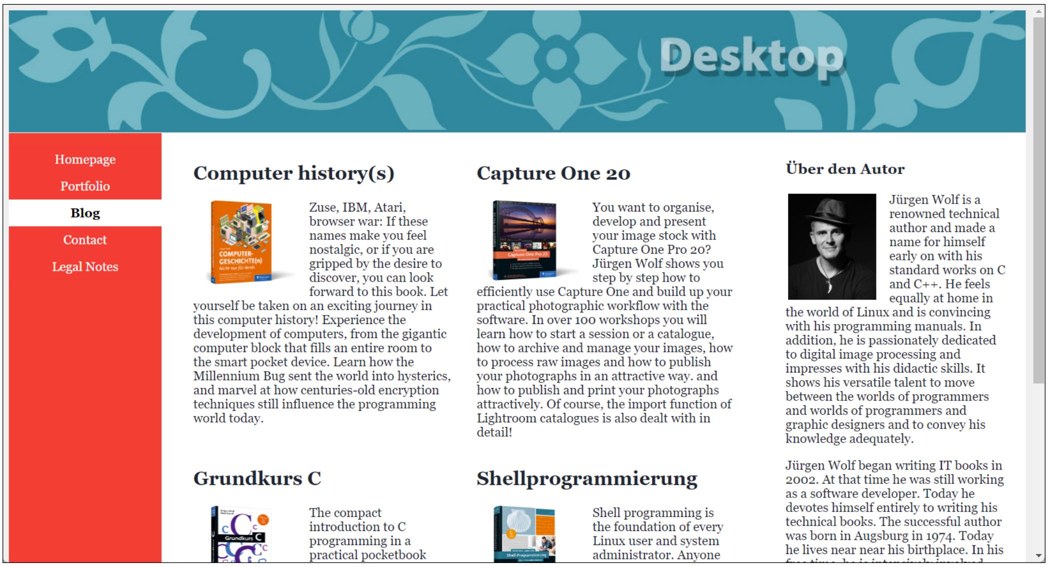
```
...
<header class="header">
<picture class="img-logo">
   <source media="(min-width: 1023px)"
           srcset="graphics/logo-desktop.jpg 1x,graphics/logo-desktop-
HD.jpg 2x">
   <source media="(min-width: 639px)"
           srcset="graphics/logo-tablet.jpg 1x,graphics/logo-tablet-HD.jpg 2x">
   <source srcset="graphics/logo-smartphone.jpg 1x,graphics/logo-smartphone-
HD.jpg 2x">
```

```
    <!-- Fallback for browsers that can't do <picture> -->
    <img src="graphics/logo.jpg" alt="Logo" class="img-logo">
</picture>
</header>
...
```

Listing 13.19 /examples/chapter013/13_3_5/index.html

Between the HTML containers `<picture>` and `</picture>`, you can use the `source` element to specify the individual image sources. Depending on the `media` attribute, corresponding images will be loaded. In the example, for displays with a width of 1,023 pixels and above (`min-width: 1023px`), the *logo-desktop.jpg* image gets loaded on ordinary displays with a single pixel density (`1x`). On the other hand, *logo-desktop-HD.jpg* will be loaded for high-resolution displays with double pixel density (`2x`).

For a screen width of less than 1,023 pixels, that is, from 1,022 pixels to 639 pixels (`min-width:639px`), either *logo-tablet.jpg* with normal pixel density or *logo-tablet-HD.jpg* with double pixel density is loaded as the image source. These image sources are intended for tablets or smaller screens. If the display width is smaller than 638 pixels, the image source *logo-smartphone.jpg* or *logo-smartphone-HD.jpg* (depending on the pixel density) for smartphones will be loaded. For web browsers that aren't able to process the `picture` element, an alternative image source is specified in the `img` element (here, *logo.jpg*), which is used instead.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Grundkurs C

The compact introduction to C programming in a practical pocketbook

Capture One 20

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish your photographs in an attractive way. and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

Shellprogrammierung

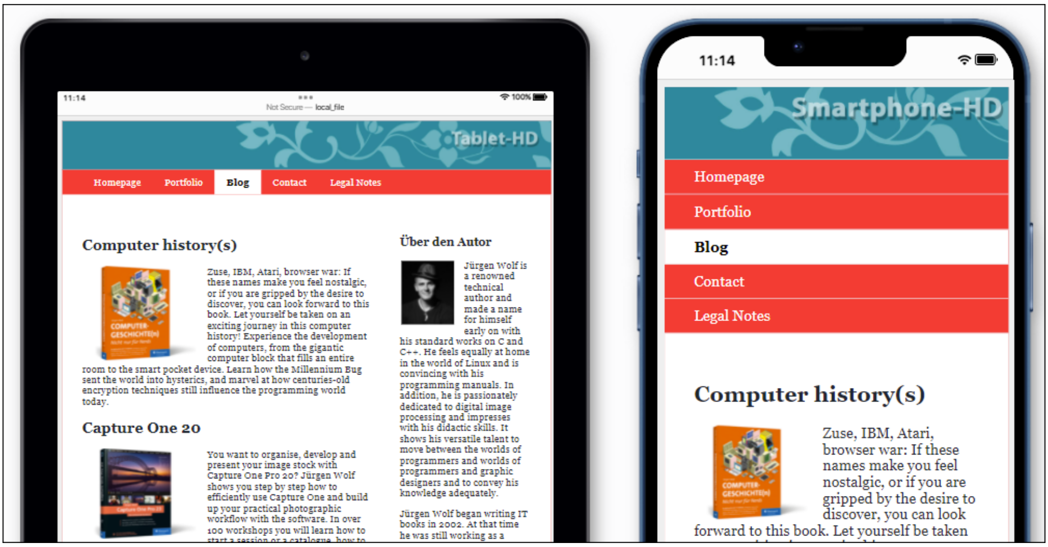
Shell programming is the foundation of every Linux user and system administrator. Anyone

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical books. The successful author was born in Augsburg in 1974. Today he lives near near his birthplace. In his

Figure 13.29 The Logo for the Desktop Version from 1,023 Pixels Onwards Was Loaded



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 20

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to

Über den Autor

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken

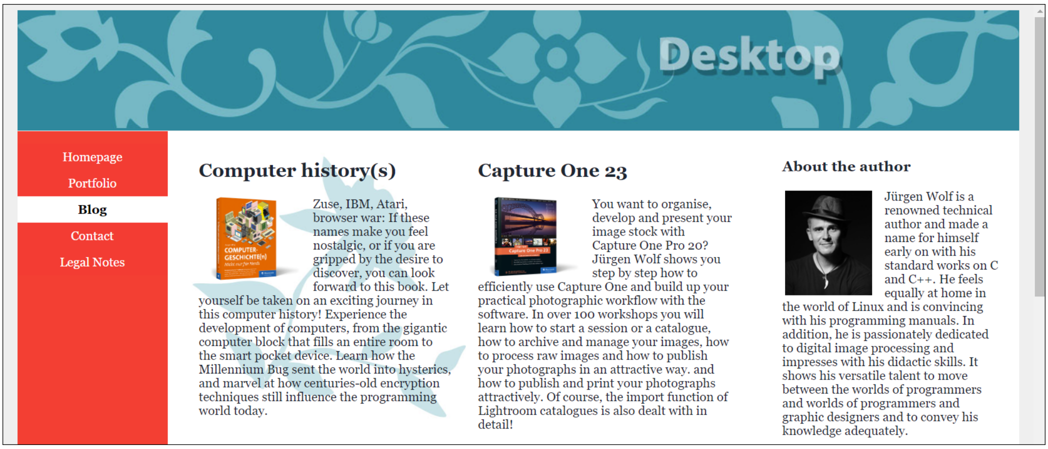
Figure 13.30 From a Display Width of 1,022 to 639 Pixels, a Smaller Image (Tablet) Is Used for the Logo, and Below 639 Pixels, the Smallest Version (Smartphone) Is Used

13.3.6 Using Area-Covering Images

If you insert background images with `background-image`, you can adjust the height and width using the CSS feature `background-size`. For example, you can use the following:

```
...
.img-background {
  background-size: 100% 100%;
  background-image: url("../graphics/background.jpg");
  background-repeat: no-repeat;
}
...
```

This way, the background image always fills the corresponding HTML element in which you use the class. You can also define an image as a background image in the `body` element for the entire HTML document. The first value represents the width, and the second value represents the height. Depending on the HTML element in which the image is used as a background, it will often be distorted, as you can see in the comparison between the desktop version in Figure 13.31 and the tablet version in Figure 13.32 with a square 500 × 500 pixel background image in an `article` element.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

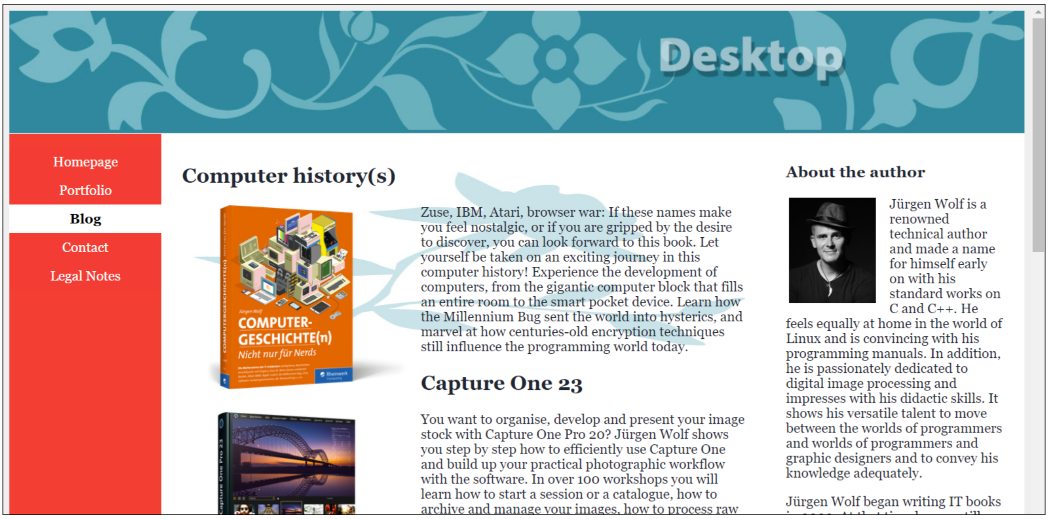
Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish your photographs in an attractive way. and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Figure 13.31 The Distortion on a Desktop Screen with “background-size: 100% 100%;” Is Still Acceptable Here



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books

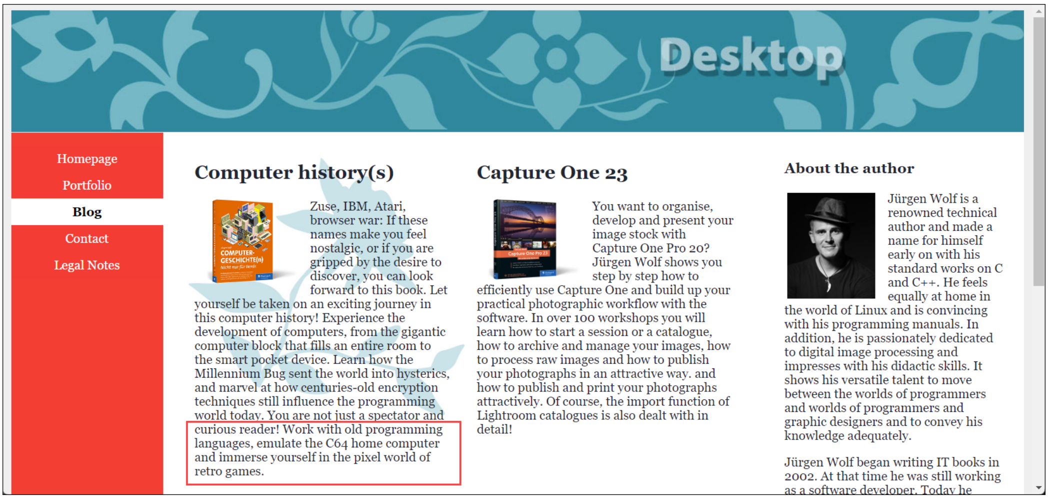
Figure 13.32 The Same Is Not True with a Smaller Screen Width: The Background Image of the First Article Is Already Distorted Significantly and Doesn’t Look Nice Anymore

If you want the image to always be stretched by 100% in width and proportionally adjusted in height, you can set the second value of `background-size` to `auto` so that the height will be adjusted to the aspect ratio, and the image no longer gets stretched out of proportion:

```
...
background-size: 100% auto;
...
```

Now the image is only stretched in width, and the height is automatically adjusted proportionally to it. When the HTML element narrows, the height value is greater than the width, and you have `background-repeat` set to `no-repeat`, a border without transitions will become visible at the bottom.

In Figure 13.33, you can see this unsightly effect with the first article: the background image is adjusted proportionally to the width, but in portrait mode, either the image is repeated or, if the repetition has been deactivated as in the example, the color of the background appears (here, a white border). In this example, it's not that dramatic because part of the background image was white anyway, but still, it doesn't look quite right. Although this makes the tablet version look better again, you'll get a similar result as in Figure 13.33 on smaller smartphones.



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish your photographs in an attractive way. and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he

Figure 13.33 A White Border Remains at the Bottom of the First <article> Element

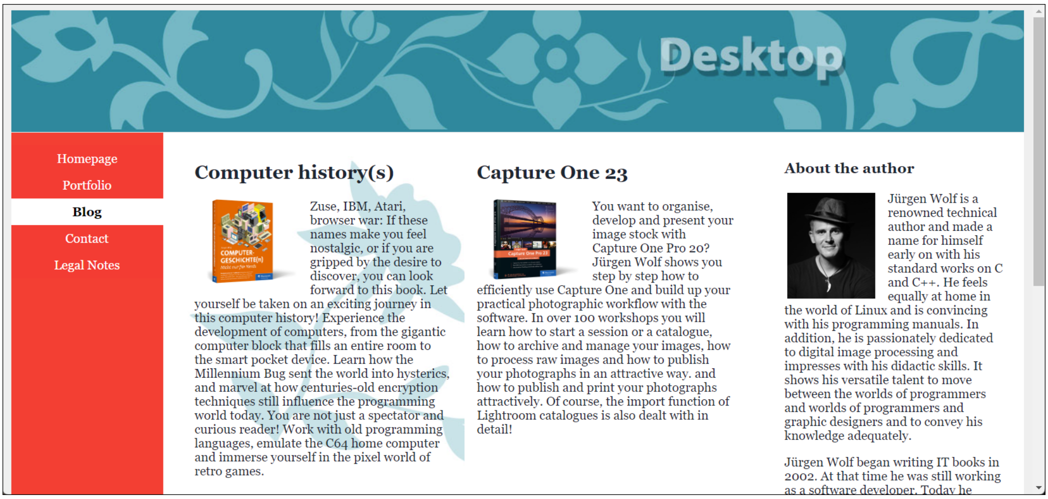
Calculating and adjusting with relative and absolute values with `background-size` rarely leads to an ideal responsive solution. What can help in this context are the two keywords `contain` and `cover`, which you can assign to the CSS feature `background-size`.

When you use `background-size: contain;`, the background image will always be displayed in its entirety. This “growing along” and “shrinking” occurs proportionally. Because the image is always visible, depending on the ratio of width and height, the background image will never fill the entire area of the screen.

The value of `cover`, on the other hand, is different. This value makes sure that the image is always displayed on the complete screen as far as possible. If the aspect ratio is different, the image will be cropped and not displayed completely. Let's look at the following solution for our example:

```
...
background-size: cover;
...
```

This seems to provide the best result, as you can see in Figure 13.34: The `cover` value was supposed to show the complete background image if possible. If it doesn't fit, it's not distorted, repeated, or truncated, but "clipped."



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your practical photographic workflow with the software. In over 100 workshops you will learn how to start a session or a catalogue, how to archive and manage your images, how to process raw images and how to publish your photographs in an attractive way. and how to publish and print your photographs attractively. Of course, the import function of Lightroom catalogues is also dealt with in detail!

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and worlds of programmers and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he

Figure 13.34 You Can Always Use "background-size: cover" to Try and Show the Entire Background Image If Possible

13.4 CSS Grid Layout

In the example you created in Section 13.2, it takes considerable effort to implement a more complex layout or redesign it. Of course, it's also possible to do it this way, but you often can't just move an element up, down, right, or left without a lot of effort.

For this purpose, *grid layouts* are the ideal solution. Whereas previously, you often created your own grid layout manually, CSS provides real design grids via CSS grid layouts, which you'll get to know a bit better on the following pages. To avoid having to create a completely new project, we're going to rewrite the known example from the previous sections for the grid layout. The complete example can be found in */examples/chapter013/13_4/css/layout.css*; the HTML document for it is located in */examples/chapter013/13_4/index.html*.

13.4.1 Creating a Grid for the Content

The principle of a CSS grid is based on the fact that you create a grid in a parent element, as you already know from responsive layout, and position the child elements in it. To do that, you must assign the `grid` value to the CSS feature `display` in the parent element and then use the features `grid-template-columns` and `grid-template-rows` to define the individual grid lines. Let's take a look at the following simple example:

```
.grid {
  display: grid;
  grid-template-rows: 150px auto auto 100px;
  grid-template-columns: 20% 20% 20% 20% 20%;
}
```

This would create a CSS grid with four rows and five columns. The first row is 150 pixels high, while the last one is 100 pixels high. The two middle rows are still adjusted according to the content using `auto`. All five columns are also 20% wide. Besides the units in percent or pixels, you can also use `em` or `fr` (e.g., `1fr` or `2fr`). The `fr` unit stands for a flexible fragment (fraction), which you can think of as the percentage of space left. Figure 13.35 shows the created CSS grid layout.

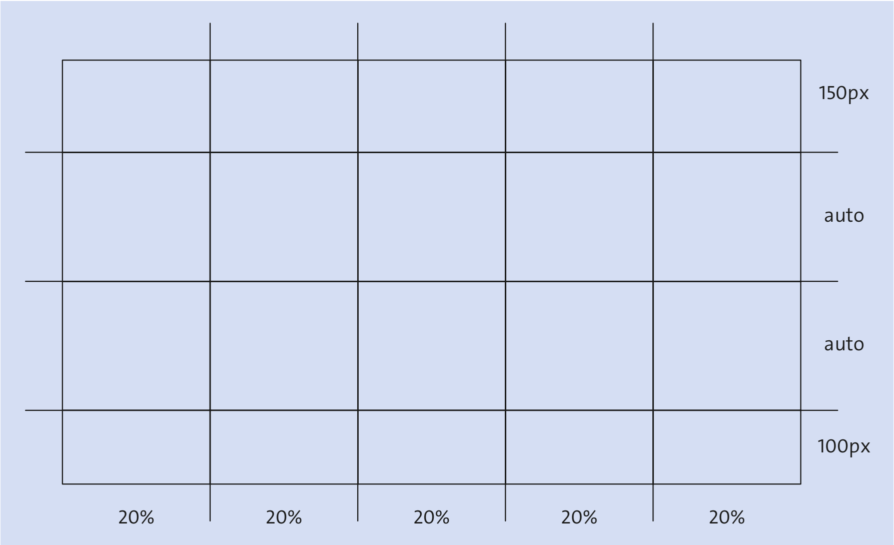


Figure 13.35 A Grid Layout with "display:grid;" Can Be Created Quickly

Because the responsive layout from Section 13.2 has been revised here and will be used for the grid layout, the grid created in Figure 13.35 is a bit too narrow. Because five columns have been defined with 20%, you can thus divide the content of the HTML elements into areas of 20%, 40%, 60%, 80%, and 100%. For the example, we want to divide the grid a bit finer in 10% steps and therefore use the following grid layout:

```
...
.grid {
  display: grid;
  grid-template-rows: 150px auto auto auto 100px;
  grid-template-columns: repeat(10, 10%);
}
...
```

Listing 13.20 /examples/chapter013/13_4/css/layout.css

The grid gets defined here even before the first layout breaks occur in the basic version. This way, you can define a grid layout with 5 rows and 10 columns. For the header and footer, we define a height of 150 and 100 pixels, respectively. The three lines in between get adjusted according to the content via `auto`. To avoid having to write `10%` 10 times, the `repeat()` function was used here. The first value of the function represents the number of repetitions of the second value—here: 10 times `10%`. Instead of the `repeat()` function, you could have written the following:

```
...
.grid {
  display: grid;
  grid-template-rows: 150px auto auto auto 100px;
  grid-template-columns: 10% 10% 10% 10% 10% 10% 10% 10% 10% 10%;
}
...
```

This sets a grid layout like the one shown in Figure 13.36.

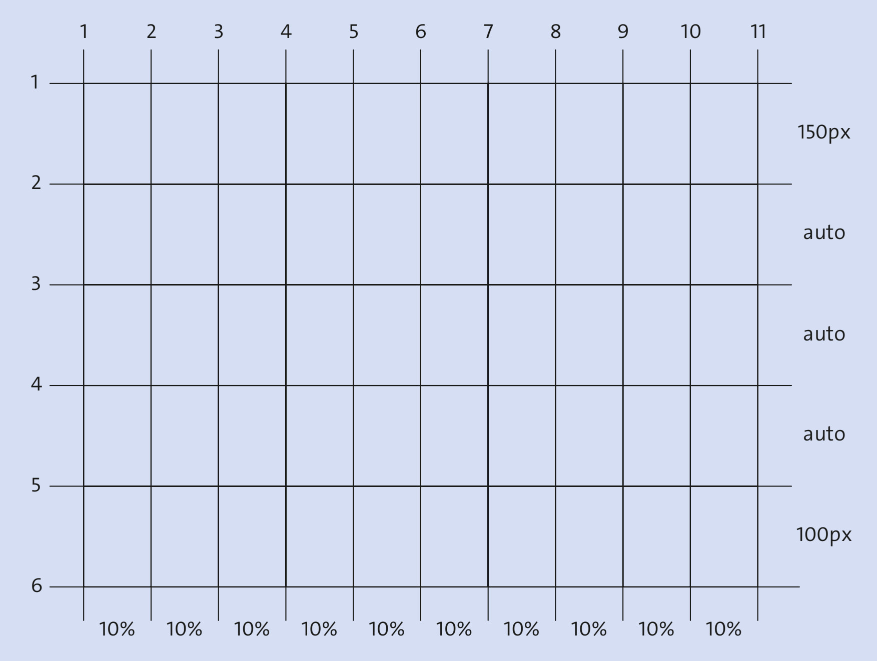


Figure 13.36 The Grid Layout for the Example

To use the grid layout with the `.grid` class selector, you must use it in the HTML document in the parent element whose child elements are positioned in this grid. In the example, we do this right after the `body` element and use a `div` element for it:

```
...
  <body>
    <div class="grid">
      <header class="header">...</header>
      <nav class="nav">...</nav>
      <main class="content">...</main>
      <aside class="aside">...</aside>
      <footer class="footer">... </footer>
    </div>
  </body>
...
```

Listing 13.21 /examples/chapter013/13_4/index.html

Inside the parent element `div` with the `grid` class, you can now position the child elements `<header>`, `<nav>`, `<main>`, `<aside>`, and `<footer>` in the grid cells shown in Figure 13.36.

13.4.2 Placing Elements in the Grid

Once you've specified the grid layout, you can easily specify where you want to place the HTML elements in the grid using the CSS features `grid-row-start` and `grid-row-end` or `grid-column-start` and `grid-column-end`. When you take a look at the grid layout in Figure 13.36, the values for `grid-column-start` and `grid-column-end` range from 1 (0%) to 11 (100%), and the values for `grid-row-start` and `grid-row-end` can be defined as 1 to 6. Thus, to make the header in the first line extend to the full width, you need to write the following:

```
...
.header {
   grid-column-start:1;
   grid-column-end:11;
   grid-row-start:1;
   grid-row-end:2;
   text-align: right;
   background-color: #07889b; /* Teal */
   color: #efefef;            /* Neutral */
   border-bottom: 1px solid #efefef;
}
...
```

Figure 13.37 shows the result of these lines.

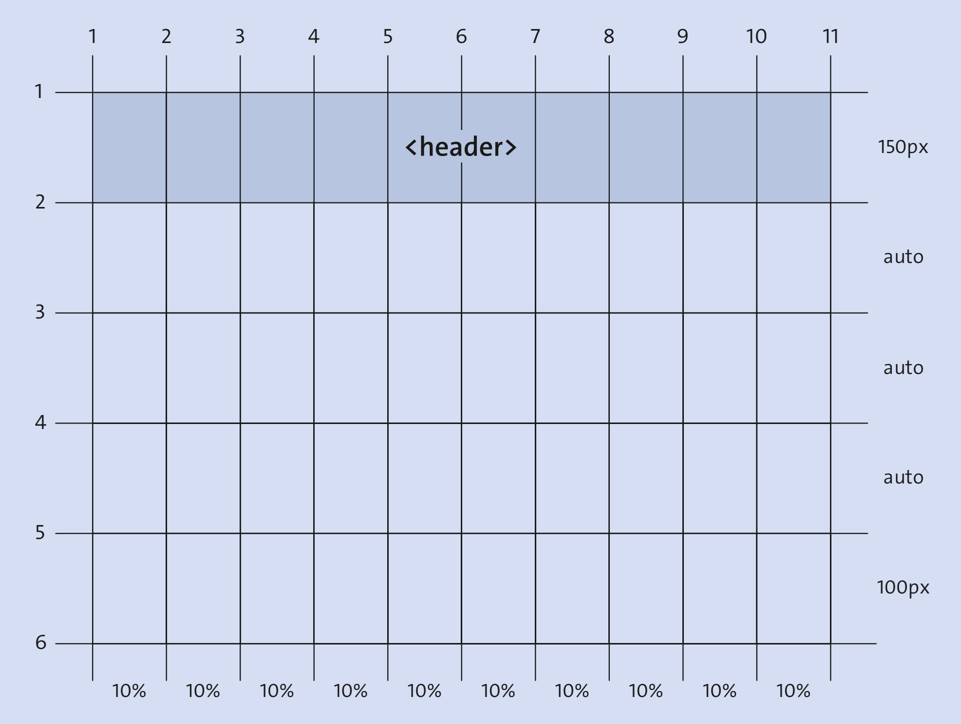

1
2
3
4
5
6
7
8
9
10
11
1
2
3
4
5
6
<header>
150px
auto
auto
auto
100px
10%
10%
10%
10%
10%
10%
10%
10%
10%
10%

Figure 13.37 The <header> Element Was Added to the Grid Layout

This way, you can now define the start and end points of the other HTML elements in the grid layout for the basic version:

```
...
.nav {
   grid-column-start:1;
   grid-column-end:11;
   grid-row-start:2;
   grid-row-end:3;
}
.content {
   grid-column-start:1;
   grid-column-end:11;
   grid-row-start:3;
   grid-row-end:4;
}
.aside {
   grid-column-start:1;
   grid-column-end:11;
   grid-row-start:4;
   grid-row-end:5;
}
.footer {
   grid-column-start:1;
   grid-column-end:11;
   grid-row-start:5;
   grid-row-end:6;
}
...
```

The result of our basic version in Figure 13.38 is now exactly the same as the mobile version you've already created in the responsive layout in Section 13.2.3.

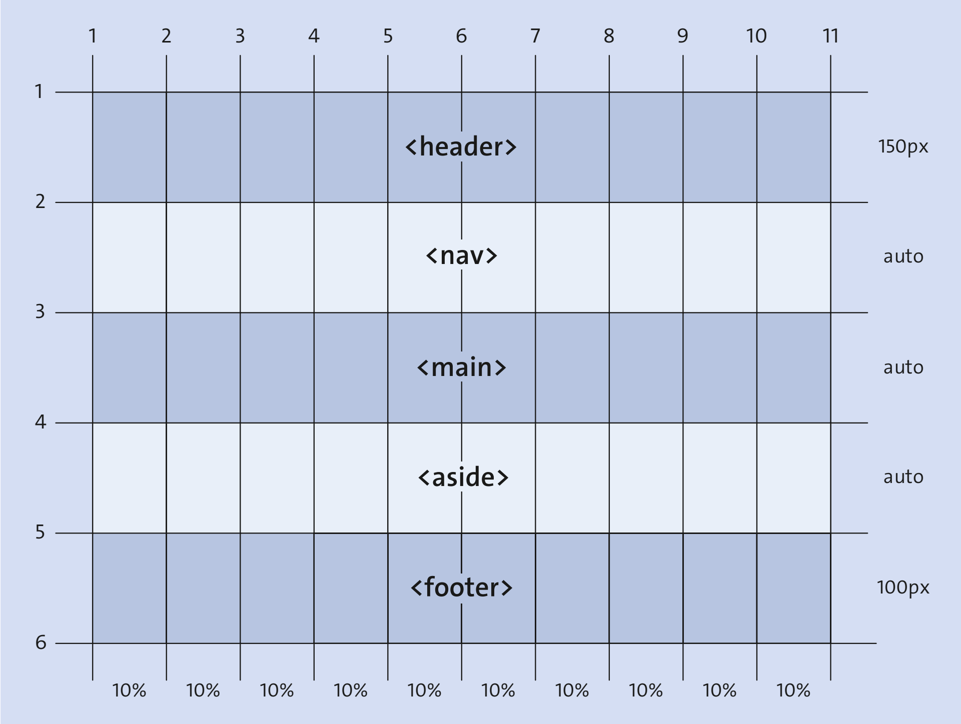


Figure 13.38 The Basic Mobile Version for Our Layout with CSS Grid

Using Shorter Notations for Placing Elements in the Grid

For the `grid-column-start` and `grid-column-end` functions, you can use the short notation `grid-column`, or instead of `grid-row-start` and `grid-row-end`, you can use the version `grid-row`. Applied to the example shown earlier with the `nav` element, you could therefore also write the following:

```
...
.nav {
   grid-column: 1 / 11;
   grid-row: 2 / 3;
}
...
```

This version corresponds to the following notation:

```
...
.nav {
   grid-column-start:1;
   grid-column-end:11;
   grid-row-start:2;
   grid-row-end:3;
}
...
```

It even gets shorter if you use the `grid-area` feature. The order of the start and end points is as follows:

```
grid-area: row-start / column-start / row-end / column-end;
```

Thus, you could also use the following third notation for positioning the `nav` element, for example:

```
...
.nav {
  grid-area: 2 / 1 / 3 / 11;
}
...
```

Placing Elements in the Next Layout Break

Starting from the basic mobile layout version, little work is now needed to respond with appropriate properties at the next layout break for the tablet version:

```
...
@media screen and (min-width: 40em) {
  .content {
    grid-column: 1 / 8;
    grid-row: 3 / 4;
  }
  .aside {
    grid-column: 8 / 11;
    grid-row: 3 / 4;
  }
...
}
...
```

With regard to our grid, you've created the layout that had been created in Figure 13.39 for the tablet version. You can see the example in use in Figure 13.40.

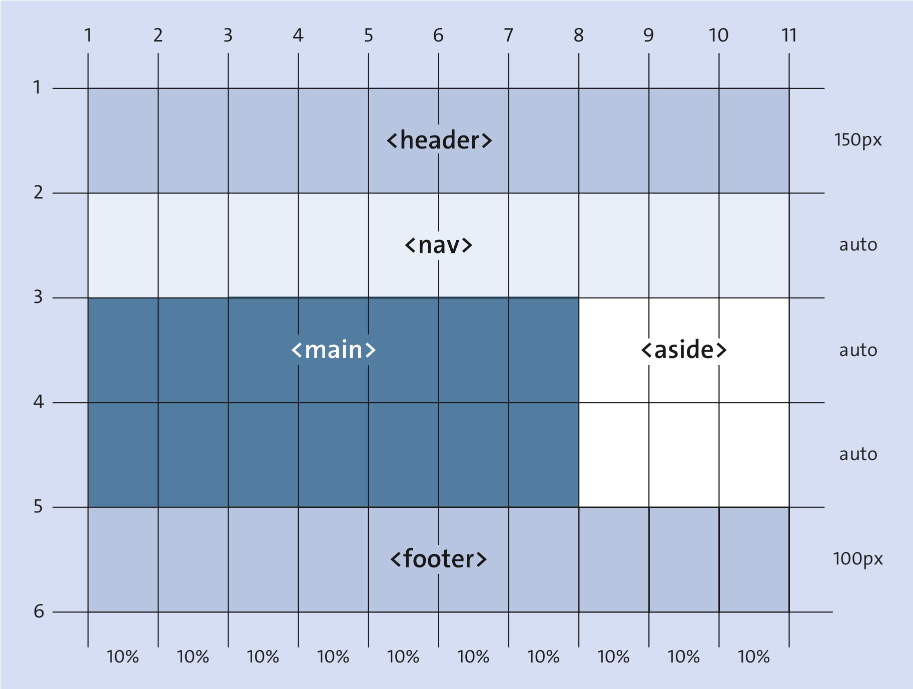
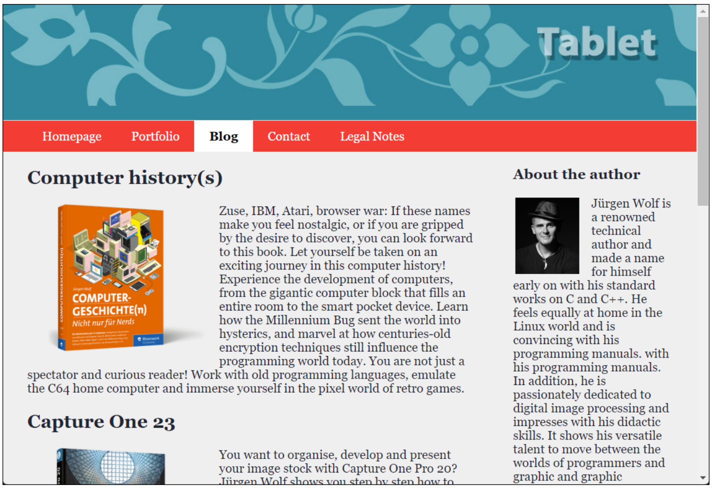


Figure 13.39 The Layout for the Tablet Version with CSS Grid



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and is convincing with his programming manuals. with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic and graphic

Figure 13.40 The Tablet Version Was Created Using a CSS Grid

Finally, another version is to be created for the desktop version:

```
...
@media screen and (min-width: 64em) {
  .content {
    grid-column: 3 / 8;
    grid-row: 2 / 4;
  }
  .aside {
    grid-column: 8 / 11;
    grid-row: 2 / 4;
  }
  .nav {
    grid-column: 1 / 3;
    grid-row: 2 / 4;
  }
...
}
```

This way, you've allocated 20% space in width for the navigation, 50% for the main content, and 30% for the sidebar. This results in the layout of the HTML elements in the grid shown in Figure 13.41.

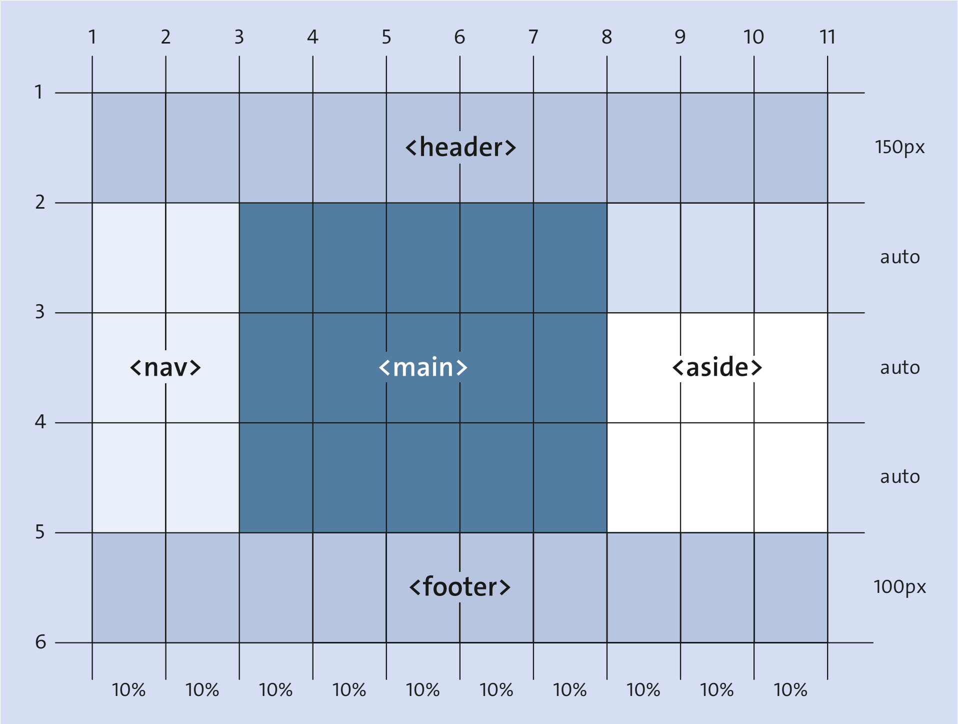
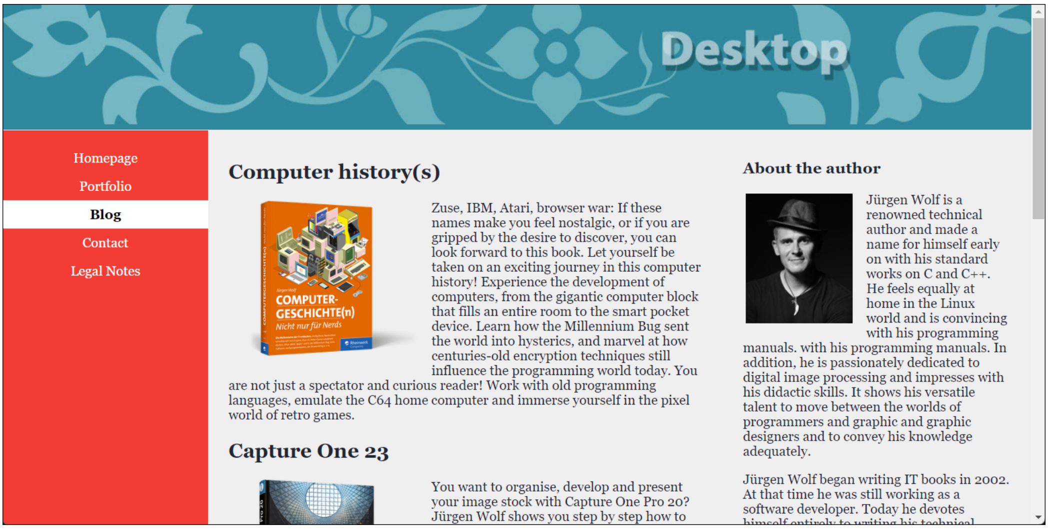

1
2
3
4
5
6
7
8
9
10
11
1
2
3
4
5
6
<header>
<nav>
<main>
<aside>
<footer>
150px
auto
auto
auto
100px
10%
10%
10%
10%
10%
10%
10%
10%
10%
10%

Figure 13.41 The Desktop Version with the CSS Grid



Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and is convincing with his programming manuals. with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic and graphic designers and to convey his knowledge adequately.

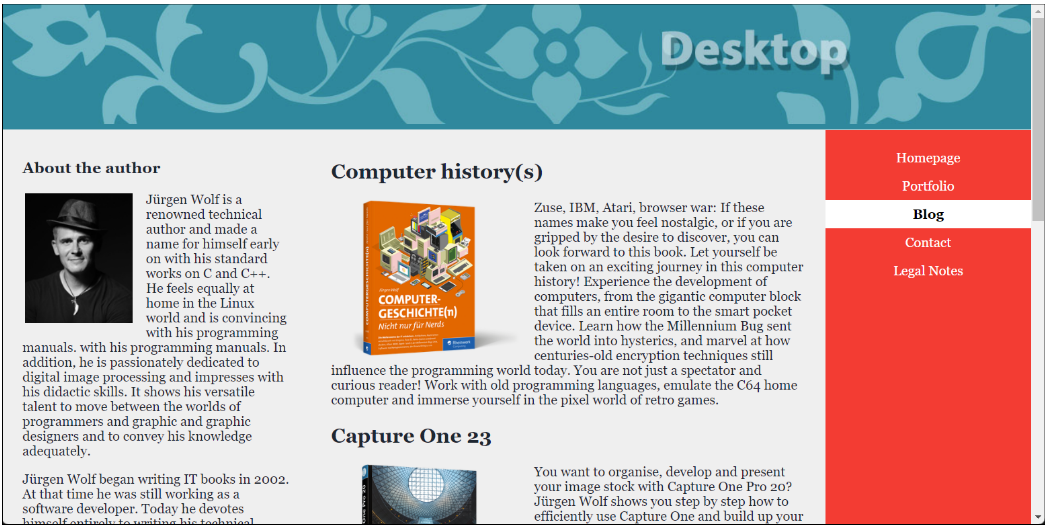
Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes

Figure 13.42 The Desktop Version with the CSS Grid in Use

13.4.3 Layout Changes Made Easy

Thanks to the simplicity of positioning the elements completely freely in the grid, it now becomes a breeze to redesign the layout. To do that, you only need to adjust the positions of the rows and columns in the grid for the HTML elements. For example, if you want to change the desktop version so that the sidebar is on the left and the navigation is on the right, you can simply change the values for `grid-column` in our example as follows:

```
@media screen and (min-width: 64em) {
  .content {
    grid-column: 3 / 8;
    grid-row: 2 / 4;
  }
  .aside {
    grid-column: 8 / 11;
    grid-row: 2 / 4;
  }
  .nav {
    grid-column: 1 / 3;
    grid-row: 2 / 4;
  }
...
}
```



About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and is convincing with his programming manuals. with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It shows his versatile talent to move between the worlds of programmers and graphic and graphic designers and to convey his knowledge adequately.

Jürgen Wolf began writing IT books in 2002. At that time he was still working as a software developer. Today he devotes himself entirely to writing his technical

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

Capture One 23

You want to organise, develop and present your image stock with Capture One Pro 20? Jürgen Wolf shows you step by step how to efficiently use Capture One and build up your

Figure 13.43 A Layout Change with a CSS Grid Can Be Done in a Few Seconds

13.4.4 Spacing between Grid Lines

If you want to add spaces between the columns or rows of a grid, you can do this in the parent element via the `grid-column-gap` or `grid-row-gap` commands or the short notation, `grid-gap`. The distances are only created between the columns. No space is added at the beginning and end of the column or row. Here's an example:

```css
.grid {
  display: grid;
  grid-template-rows: 150px auto auto auto 100px;
  grid-template-columns: repeat(10, 10%);
  grid-row-gap: 15px;
  grid-column-gap: 10px;
  /* or as short notation: grid-gap: 15px 10px; */
}
```

Aligning Elements in the CSS Grid

You can also specify the horizontal and vertical alignment of the elements in the parent element using the CSS features `align-items` for vertical behavior and `justify-items` for horizontal behavior. The values `start`, `end`, `stretch`, and `center` are available for that. For an individual alignment of a single grid cell, on the other hand, you can use `align-self` and `justify-self`. The values `start`, `end`, `stretch`, and `center` are also available here.

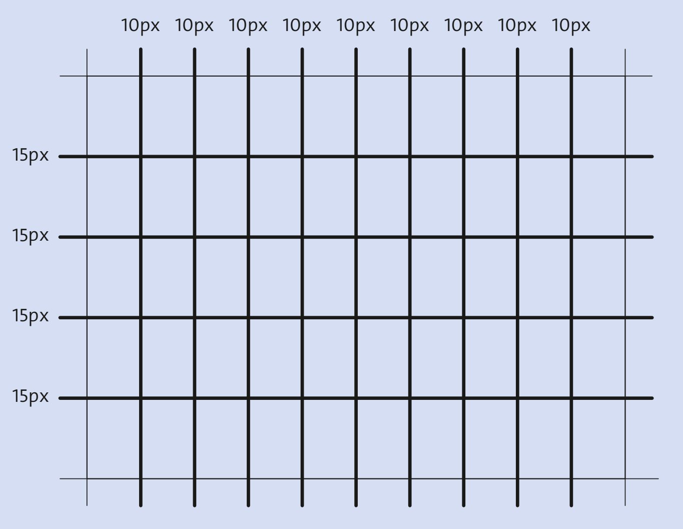
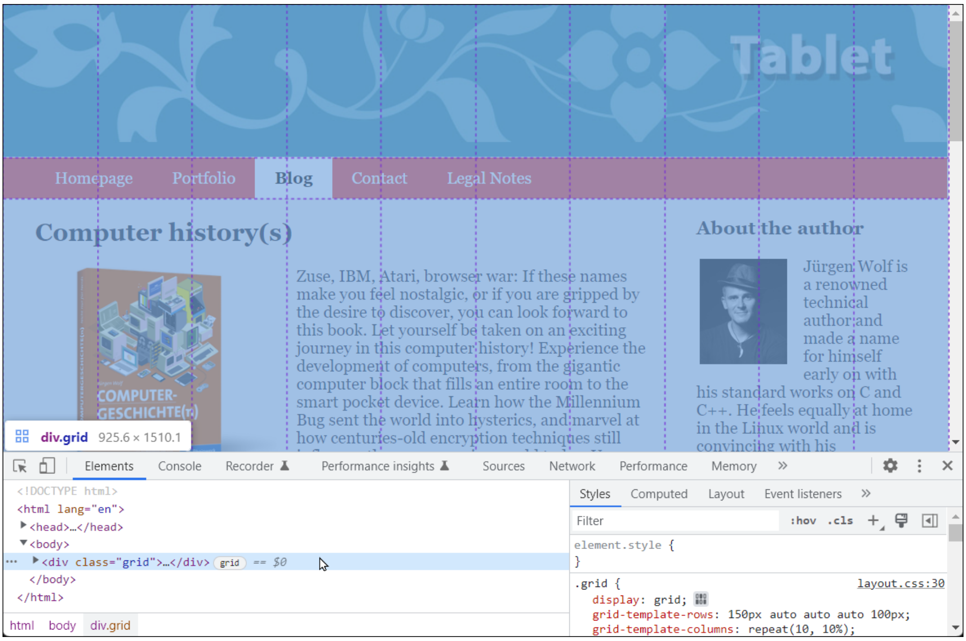

10px
10px
10px
10px
10px
10px
10px
10px
10px
15px
15px
15px
15px

Figure 13.44 Adding Spacing between the Columns of a CSS Grid

13.4.5 Checking the Grid in the Web Browser

CSS grids are also worth taking a look in the developer tools (e.g., with `Ctrl`+`Shift`+`I`) of the web browser, which usually makes them visible when you select the corresponding HTML element used as a container for the grid. This visual view is very helpful when you look for errors or check the grid layout.


Tablet
Homepage
Portfolio
Blog
Contact
Legal Notes
COMPUTER-GESCHICHTE(r)
div.grid 925.6 × 1510.1
Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory
Styles
Computed
Layout
Event listeners
Filter
:hov
.cls
html
body
div.grid

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the Linux world and is convincing with his

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    ▶<div class="grid">…</div> grid == $0
   </body>
</html>
```

```
element.style {
}
.grid {                                                layout.css:30
   display: grid;
   grid-template-rows: 150px auto auto auto 100px;
   grid-template-columns: repeat(10, 10%);
```

Figure 13.45 The Grid Is Also Displayed in the Developer Tools of the Web Browser

13.5 Changing the Behavior of HTML Elements Using “display”

You’ve used the CSS feature `display` many times in the examples, so we’ll dedicate a few more paragraphs to it for a brief description. As you’ve seen several times by now, you can use the CSS feature `display` to change the behavior of an HTML element when it displays in the web browser. For each HTML element, a box is specified that describes the behavior of the element. Even simple HTML elements inside a text line such as `<em>` or `<a>` are boxes, and you can change their default behavior via `display`. This has already been described in greater detail.

You can change the behavior of an HTML element such as `<p>` by using `display: inline;` which will no longer executes a line break or paragraph break. Conversely, you can change the behavior of an element such as `<a>` by using `display: block;` so that it performs a line break or paragraph break. In addition to the common use of `display: block;` and `display: inline;`, `display: none;` can also be used; this way, you can hide an element so that it no longer takes up any space in the HTML document.

13.5.1 “display: block”, “display: inline”, and “display: inline-block”

`display: block;` allows you to display an element as a block that contains a line break. This feature is often used in combination with `display: inline;`, for example, elements for navigation are displayed either with `display: block;` with a line break between them, or with `display: inline;` in the same line without a line break from left to right.

For demonstration purposes, the following CSS features will be applied to several `p` elements in the HTML document:

```
p {
  display: block; /* Not necessary here, since <p> is display:block anyway */
  width: 150px;
  border: 1px solid black;
  background-color: white;
  padding: 1em;
}
```

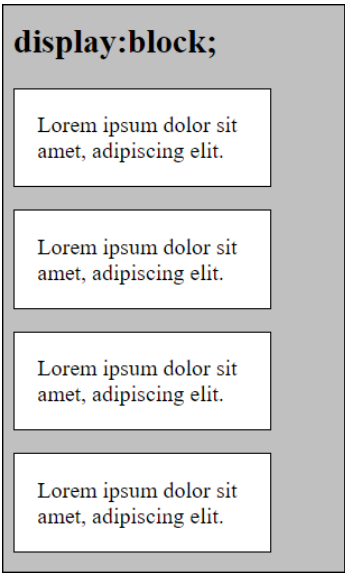
Listing 13.22 /examples/chapter013/13_5_1/index.html

You could do without `display: block;` in this case because the `p` element is a block element anyway. These lines of CSS are now to be applied to four paragraph texts with the `p element`. An example of this is shown in Figure 13.47.

The next example is to use `display: inline;` instead of `display: block;`:

```
p {
  display: inline;
  width: 150px;
  border: 1px solid black;
  background-color: white;
  padding: 1em;
}
```

Listing 13.23 /examples/chapter013/13_5_1/index2.html



display:block;

Lorem ipsum dolor sit amet, adipiscing elit.

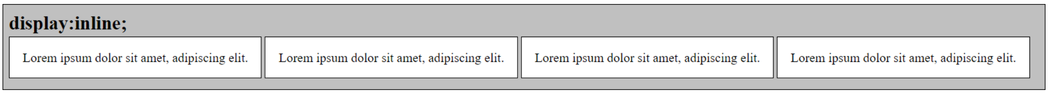
Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Figure 13.46 The Behavior You Know from the <p> Element

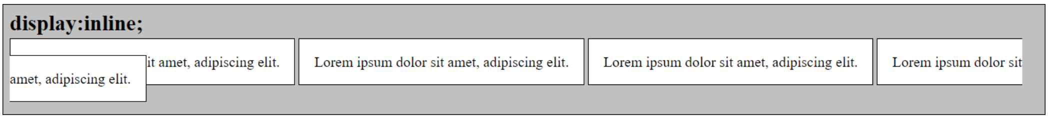
In Figure 13.47, you can already see that due to `display: inline;`, the specification of `width` was ignored and has no effect here. Although you can specify inner and outer spacing and borders as usual, these specifications also have no effect on the line height. Thus, an `inline` box absorbs only the width the content requires. In Figure 13.48, you can see that an `inline` box can also extend across multiple lines, which isn't very nice to look at in this case.



display:inline;

Lorem ipsum dolor sit amet, adipiscing elit. Lorem ipsum dolor sit amet, adipiscing elit. Lorem ipsum dolor sit amet, adipiscing elit. Lorem ipsum dolor sit amet, adipiscing elit.

Figure 13.47 The Behavior of the <p> Elements Was Set to “display: inline;”



display:inline;

it amet, adipiscing elit. Lorem ipsum dolor sit amet, adipiscing elit. Lorem ipsum dolor sit amet, adipiscing elit. Lorem ipsum dolor sit

amet, adipiscing elit.

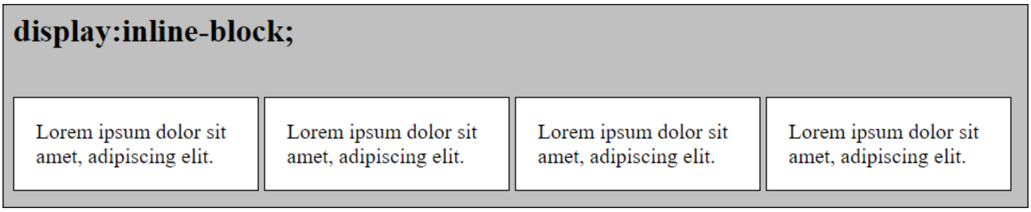
Figure 13.48 “inline” Boxes Can Also Extend across Multiple Lines

Let’s now use the same example with `inline block`:

```
p {
  display: inline-block;
  width: 150px;
  border: 1px solid black;
  background-color: white;
  padding: 1em;
}
```

Listing 13.24 /examples/chapter013/13_5_1/index3.html

An `inline-block` box initially behaves like an `inline` box and runs across one line (see Figure 13.49). Unlike the `inline` box, however, an `inline-block` box doesn’t continue in the next line but is moved to the next line, similar to `float`, when the box no longer fits in the screen width, as you can see in Figure 13.50. Unlike `inline` boxes, what’s also taken into account for `inline-block` boxes is the width that you specify via `width`.



display:inline-block;

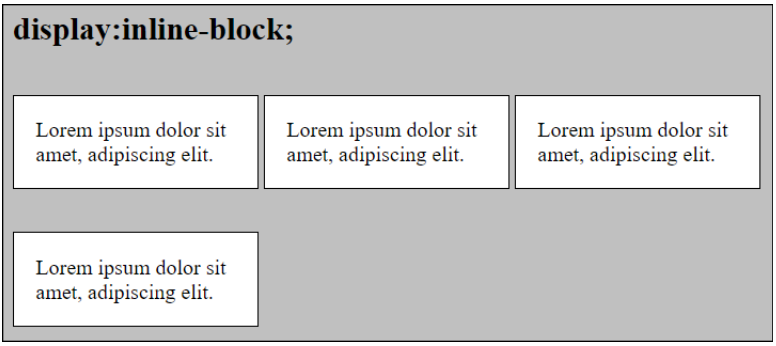
Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Figure 13.49 Here, I’ve Set the Behavior of the <p> Elements to “display:inline-block;”



display:inline-block;

Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Lorem ipsum dolor sit amet, adipiscing elit.

Figure 13.50 An “inline-block” Box Can’t Be Split across Multiple Lines

13.5.2 Hiding Elements Using “display:none”

`display: none;` allows you to easily hide elements so that they are no longer visible in the document. The web browser doesn’t create a box for such elements, and all positioning statements are ignored. Besides `display: none;`, you can also hide elements using `visibility: hidden;`. Unlike `display: none;`, however, the box gets preserved, and the element retains its effects on subsequent elements. `visibility: hidden;` allows you to make the element completely transparent.

13.5.3 Further Values for "display"

There are a few more values you can use to change the behavior of elements with `display`. With `display: flex;`, you have an alternative model for positioning elements in rows and columns. `display: grid;`, on the other hand, is even more flexible and allows you to create a more complex layout grid in which you can place any element you want. I've already described `flex` and `grid`.

Another way to change the behavior of elements with `display` is `table` and other `table-*` features. This allows you to arrange elements as you would in a table and, in practice, theoretically create a layout for a website. But there's a much better alternative for that: CSS-Grid (`display:grid`), which enables you to create a responsive layout with only a few CSS rules.

Another value to mention here is `list-item`, which you can use to represent the item as a list. It creates two boxes for one element. One box is used for the list item and the other box for the list element.

There are yet other values available for `display`, but those are used rather rarely or were either not implemented properly or never. An overview of the available `display` values can be found at *https://developer.mozilla.org/de/docs/Web/CSS/display*.

13.6 Calculations Using CSS and the "calc()" Function

Sometimes, it can be useful to have the individual specifications calculated and displayed depending on the media feature. In CSS, you can use the `calc()` function for this purpose, which allows you to perform the basic arithmetic operations, addition (+), subtraction (-), multiplication (*), and division (/). When doing that, you can also mix units and, for example, multiply pixels by a percentage. It's also important to know that the plus and minus must be preceded and followed by a space, which isn't necessary for multiplication and division operations.

To demonstrate the use of `calc()`, we'll create a simple layout grid with eight sections. In the standard version for screens larger than 640 pixels, four sections are to be distributed in a row (4 × 2). For screens that are less than 640 pixels wide, a layout break is supposed to be applied, with only two columns per row (2 × 4). At each layout break the number of columns is recalculated to fit the width. For this purpose, the `.column` class is used in the following example. If the screen width is less than 480 pixels, one column per line will be used (1 × 8). However, the example is only intended to demonstrate `calc()` in practice, and should in no way be taken as a recommendation for creating layouts. Flexboxes and CSS grids are a much better solution for that. Here’s the corresponding example:

```
...
.column {
  float: left;
  padding: 10px;
  width: 90%; /* For browsers without calc() support */
  width: calc(100% / 4);
}

@media screen and (max-width: 40em) {
  .column {
    width: calc(100% / 2);
  }
}

@media screen and (max-width: 30em) {
  .column {
    width: 100%;
  }
}
```

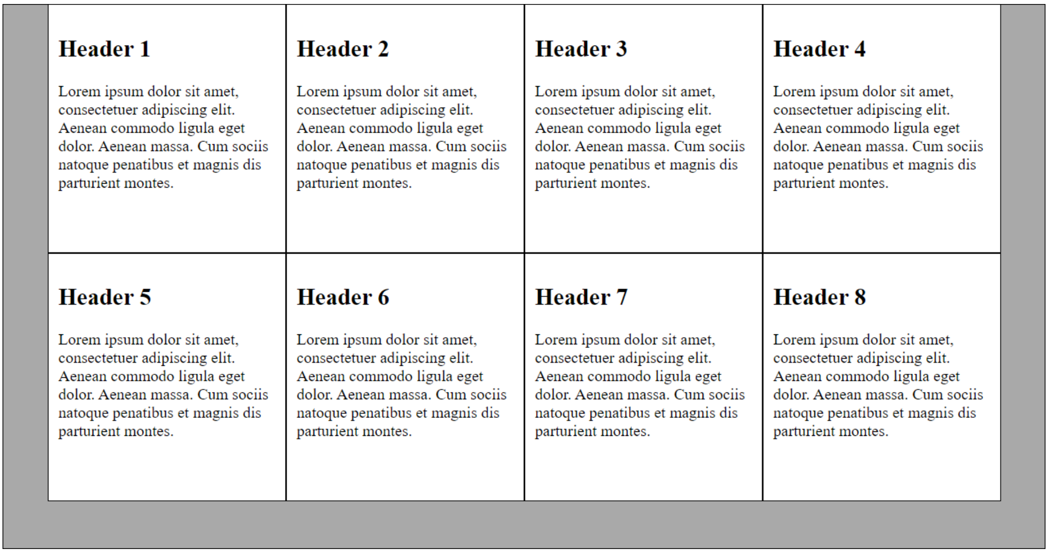
Listing 13.25 /examples/chapter013/13_6/css/layout.css

The corresponding HTML code looks like the following:

```
...
<section class="column">…</section>
<section class="column">…</section>
<section class="column">…</section>
<section class="column">…</section>
<section class="column">…</section>
<section class="column">…</section>
<section class="column">…</section>
<section class="column">…</section>
...
```

Listing 13.26 /examples/chapter013/13_6/index.html

In the following figures, you can see how `calc()` is used to calculate a new layout grid at each layout break. Only the `.column` class is used here.



Header 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 4

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 5

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 6

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

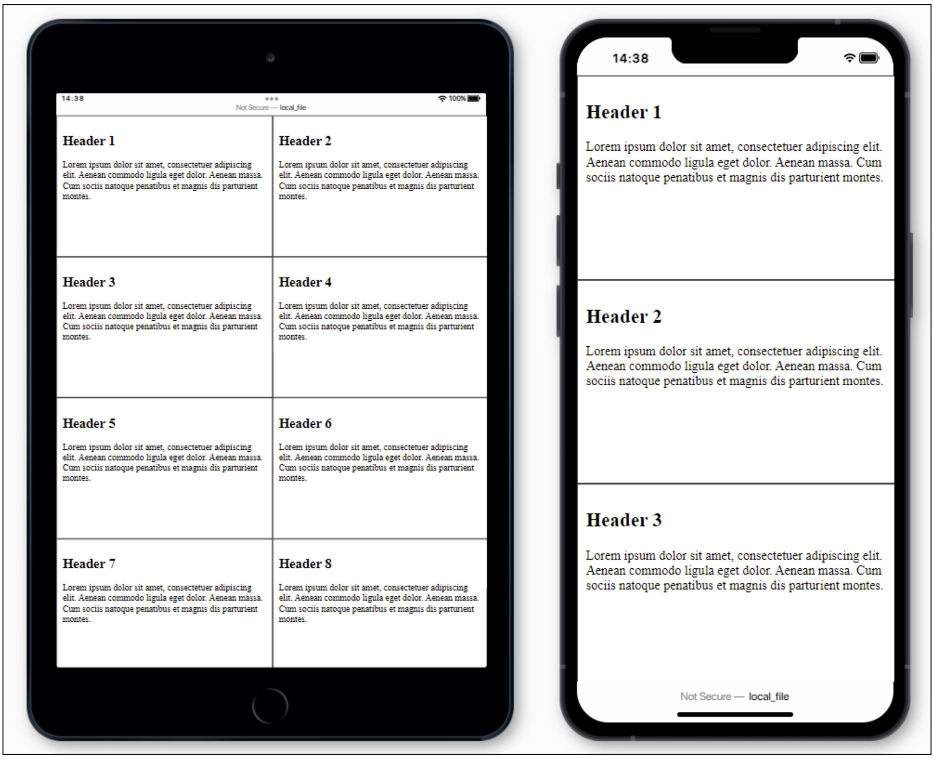
Header 7

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 8

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Figure 13.51 A Four-Column Layout with "width: calc(100% / 4);" for a Viewport of More Than 640 Pixels



Header 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 4

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 5

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 6

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 7

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 8

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Header 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes.

Figure 13.52 A Two-Column Layout with "width: calc(100% / 2);" for a Viewport of Less Than 640 Pixels (Left) and a Single-Column Layout with "width: 100%;" for the Viewport of Less Than 480 Pixels (Right)

Of course, the areas of application for `calc()` go far beyond this example. For instance, it can also be used for absolute positioning on the screen or automatic adjustment of individual elements within the parent element. You can even calculate a flexible adjustment of the font size using `calc()`.

13.7 Summary

Although I'd like to provide even more detail on the topic of responsive layout using CSS, this book isn't a pure CSS manual or a book on responsive web design. But you now know the basics about responsive web design and are familiar with media queries. You also know useful building blocks that enable you to get creative and design simple layouts.

A few words about the examples in this chapter, which you should understand as simple suggestions only. The possibilities of creating layouts with CSS are extremely diverse and in practice often more elaborate. But that's the beauty of web development: you can be creative and create something unique yourself. To create something special, you need to gain experience, try as much as possible, and work your way up step by step. There's seldom a silver bullet along the lines of "Do it this way, and it will be perfect!" because it all depends on the needs (and also the knowledge) of each individual and, of course, on what you want to create.

In addition, over the years, the technique of how to create CSS layouts changes. Until recently, only the `float` feature was used for layouts, but now, thanks to broad browser support, the CSS grid layout and the flexbox are finally gaining more and more acceptance as the current standard for layout creation. I still demonstrated the `float` feature here for introducing and creating a layout, and later showed you with the CSS grid layout how much easier it is to create and especially rearrange a layout with it.

Chapter 14

Styling with CSS

Welcome to the pleasure dome! Now that you've familiarized yourself with the important basics of CSS, such as selectors, the box model, positioning, and layout, you can turn your attention in this chapter to things such as website design—or, more simply—the CSS elements you can use to make websites more beautiful.

This chapter covers other ways to make websites more beautiful or readable that haven't been described up to now. The main focus is clearly on working with text at first, as text is usually the most important thing on most websites. In addition, you may want to design elements such as lists or tables with CSS. Likewise, I'll briefly describe the design of images and graphics with CSS. You'll also learn newer options such as moving and rotating or animating elements using CSS. At the end of the chapter, I'll briefly describe how you can design HTML forms.

The following topics are covered in this chapter:

- Designing texts with CSS
- Styling ordered and unordered lists with CSS
- Making tables more beautiful with CSS
- Designing graphics and images with CSS
- Transforming HTML elements with CSS
- Creating smooth transitions with CSS
- Styling HTML forms with CSS

14.1 Designing Texts with CSS

The purpose of most websites on the internet is to convey information. Usually, the flow of information on websites consists of text, images, and videos. The most important type of information flow on the web is text. CSS provides an impressive amount of CSS features for this purpose, which you can use to design or customize texts for websites. I'll describe these CSS features in more detail in the following sections.

14.1.1 Selecting Fonts via “font-family”

You can use the CSS feature `font-family` to select the font for the text within an element. As the value for this CSS feature, you can pass the name of the font you want to use to format the text within the HTML element, for example:

```
body { font-family: Arial; }
```

This sets the font between `<body>` and `</body>` to Arial.

A prerequisite for the corresponding font to be used for display is that it must be installed on the local system of a visitor to the website. In the case of the Arial font, this is probably pretty much the case. Nevertheless, you can specify several alternatives separated by commas—called a *font stack*. Here’s an example:

```
body { font-family: Arial, Verdana, sans-serif; }
```

In this case, the Arial font is used between the HTML elements `<body>` and `</body>`. If the visitor doesn’t have this font installed on his system, the web browser can use Verdana as an alternate font. If that font also isn’t available on their system, you instruct the web browser to select any sans-serif font on the system and use it to display the text.

The list can be as long as you want, and the web browser will use the first font installed on the system. Fonts that contain a space in their name must be specified between quotation marks (e.g., `"Courier New"`).

If No Suitable Font Is Available

If no specified font from `font-family` is available on the system, the default font of the web browser will be used.

Overview of Generic Fonts

To be on the safe side, it’s recommended to specify a generic font (or font class) in a list of different fonts at the end. There are five different generic fonts listed in Table 14.1. In Figure 14.1, you can see the different font classes printed for better distinction.

Font Class	Meaning	Known Examples

`serif`	In serif fonts, you’ll find small fine lines or tick marks at the end of the letter stroke across the base direction.	`Times`
		`Times New Roman`
		`Georgia`
		`Bookman`

Table 14.1 Various Generic Font Classes

Font Class	Meaning	Known Examples

`sans-serif`	These are sans-serif fonts where the end of the stroke is straight.	`Arial`
		`Verdana`
		`Helvetica`
		`Lucida`

`monospace`	These are fonts with a fixed width, where all letters have the same width.	`Courier`
		`Courier New`
		`Andale Mono`
		`Fixed`

`cursive`	The name is somewhat confusing because these are fonts that are meant to give the impression of a cursive script.	`Comic Sans MS`
		`Florence`
		`Parkavenue`
		`Monotype Corsiva`

`fantasy`	These are often decorative ornamental fonts that can be used for creative purposes and are less suitable for entire passages of text.	`Impact`
		`Haettenschweiler`
		`Oldtown`
		`Brushstroke`

Table 14.1 Various Generic Font Classes (Cont.)

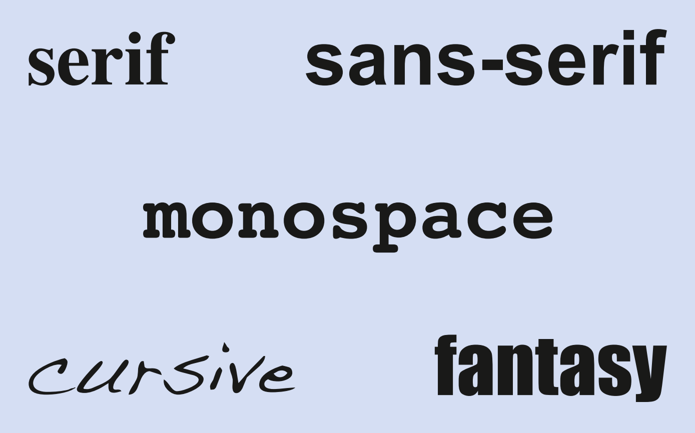


Figure 14.1 The Five Different Generic Fonts: “serif”, “sans-serif”, “monospace”, “cursive”, and “fantasy”

Of course, you could simply specify just a generic font such as a `sans-serif`:

```
body { font-family: sans-serif; }
```

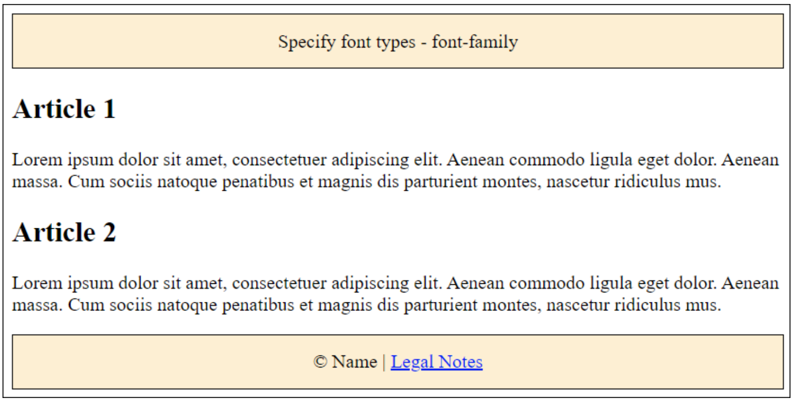
However, it can’t be predicted with certainty which sans-serif font will then be used to display text.

Inheritance of Fonts

Fonts are inherited by the subordinate elements as long as no custom font has been written in the subordinate elements. Often, therefore, a font is defined for `<body>` that applies to the entire document, for example:

```
body { font-family: Arial, Verdana, Helvetica, sans-serif; }
```

In Figure 14.2, you can see an HTML document */examples/chapter014/14_1_1/index.html* without `font-family` for the `body` element, while in Figure 14.3, there's one with the CSS feature `font-family` for the `body` element.



Specify font types - font-family

Article 1

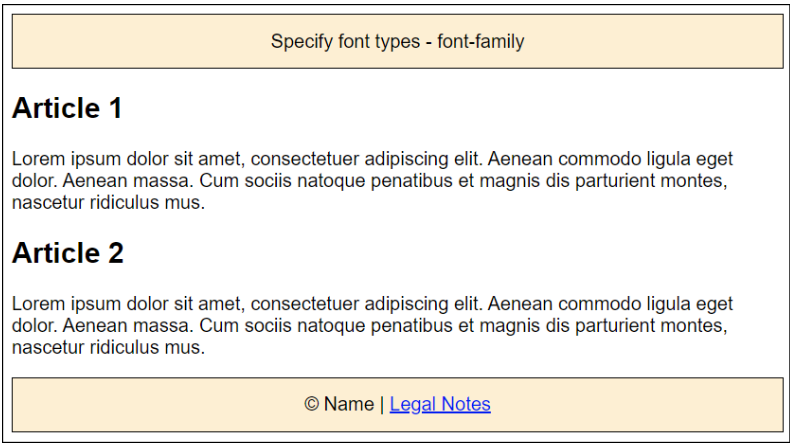
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

© Name | Legal Notes

Figure 14.2 An HTML Document with the Default Font of the Web Browser



Specify font types - font-family

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

© Name | Legal Notes

Figure 14.3 The Same Document Again, but Now with the CSS Feature "font-family": Sans-Serif Font (Here, Arial) Was Used

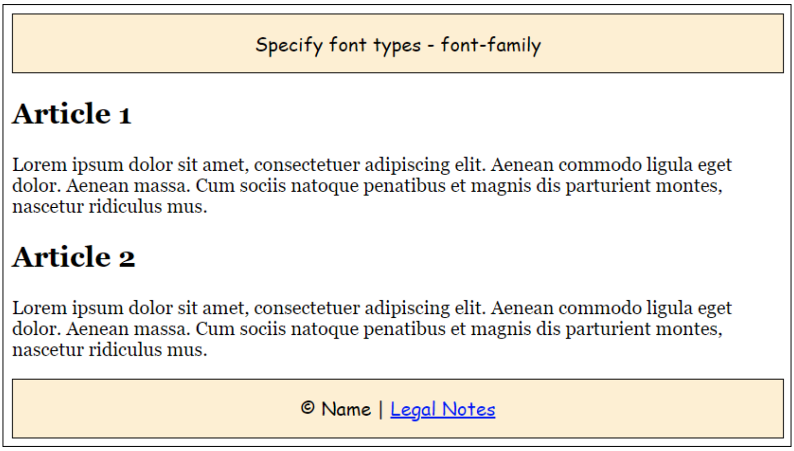
I've slightly extended this example for demonstration purposes:

```
...
body { font-family: Arial, Verdana, Helvetica, sans-serif; }
```

```
.footer, .header {
    background-color: papayawhip;
    border: 1px solid black;
    padding: 2% 2%;
    text-align: center;
    font-family: cursive;
}

.article { font-family: Georgia, Times, serif; }
...
```

Listing 14.1 /examples/chapter014/14_1/css/style.css

We first use the `body` type selector to specify a sans-serif font such as Arial, Verdana, Helvetica, or even a generic font for the entire document. This font is used as the default font if no other font is specified for an element. Then, for the `footer` and `header` elements, we let the system select a cursive type with the generic font class `cursive`, so the result here will probably look different on different computers. For the text in the `article` elements themselves, we use a serif font such as Georgia, Times, or any other serif font available on the system. The results are displayed in Figure 14.4.



Specify font types - font-family

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

© Name | Legal Notes

Figure 14.4 Multiple Different Fonts in Use

Number of Different Fonts

In practice, you should keep the number of different fonts on a web page rather low. Too many fonts won't necessarily make a website look better. A good guideline should be to use no more than three or four fonts. However, this also depends on the type of website.

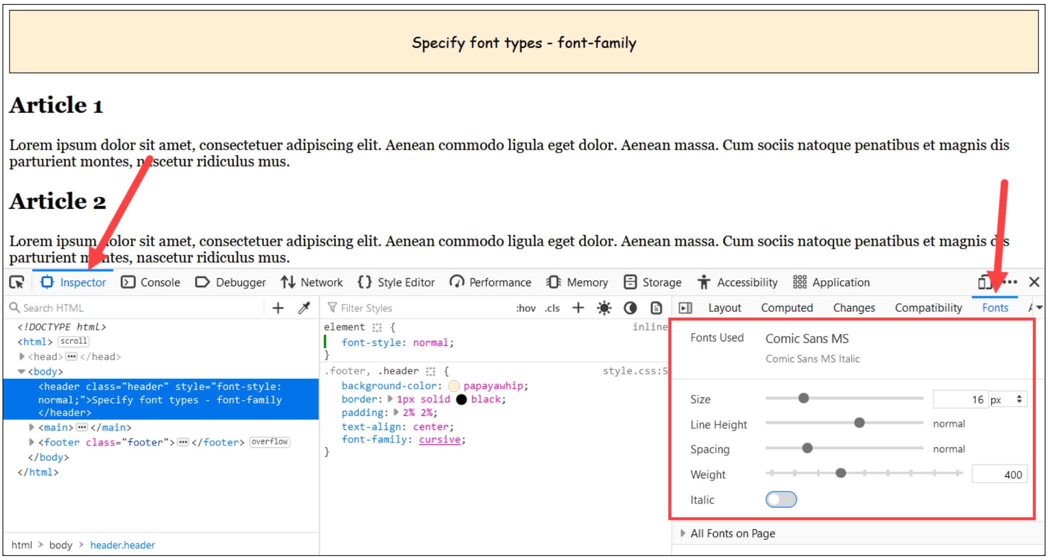
Benefits and Drawbacks of “font-family”

The drawback of using `font-family` to select the font is that a font must be installed on the visitor’s computer. So you’re quite limited in the choice of fonts. Mostly common fonts such as Times, Times New Roman, Georgia, Helvetica, Arial, or Verdana are used.

The use of `font-family` does have one advantage: you don’t have to worry about font licenses because you don’t share the font.

Analyzing Fonts in Firefox

At this point, I’d like to discuss the **Fonts** tab, which you can find in the Developer Tools in the Firefox web browser. In addition to the default settings of the web browser, you can also use it to determine the fonts of other websites if you particularly like one of them. You can also make adjustments to the font size, line height, character spacing, or stroke width here to see in the browser what it would look like with different settings. The matching values will then be displayed inline in the HTML element.



Specify font types - font-family

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Search HTML
Filter Styles
:hov
.cls
Layout
Computed
Changes
Compatibility
Fonts
html › body › header.header
All Fonts on Page

```
<!DOCTYPE html>
<html> scroll
  <head>⋯</head>
  <body>
    <header class="header" style="font-style:
    normal;">Specify font types - font-family
    </header>
    <main>⋯</main>
    <footer class="footer">⋯</footer> overflow
  </body>
</html>
```

```
element {                                  inline
  font-style: normal;
}
.footer, .header {                    style.css:5
  background-color: papayawhip;
  border: 1px solid black;
  padding: 2% 2%;
  text-align: center;
  font-family: cursive;
}
```

Fonts Used: Comic Sans MS
Comic Sans MS Italic

Size: 16 px
Line Height: normal
Spacing: normal
Weight: 400
Italic

Figure 14.5 You Can Analyze and Change the Font Used on a Web Page in Firefox, Which Makes the Effects Visible in the Browser Window

14.1.2 Providing Fonts via Web Fonts: “@font-face”

`@font-face` allows you to use fonts that aren’t installed on visitors’ computers. To do this, you just need to specify a path from where the font can be downloaded.

To add fonts to a website using the `@font-face` rule, you need the following:

- **`font-family`**
  Specify the name of the font. You can then pass this name as a value for the `font-family` feature.
- **`src`**
  Set the path or link to the font file. You can also provide different versions of the font.
- **`format`**
  Specify the file format in which the font is available.

Here's a theoretical example of how you can include such a downloadable font for a website:

```
@font-face {
   font-family: 'A font name';
   src: url('path/to/my/font.ttf') format('truetype');
}
```

Again, you can use these web fonts in CSS via the CSS feature `font-family`:

```
body { font-family: 'A font name', Times, Georgia, serif; }
```

As a fallback solution, you can specify a list of alternate fonts if the font (here, `'A font name'`) couldn't be downloaded and used.

Longer Loading Time

Logically, adding additional resources to your website, as you did here with the web fonts, also means that the loading time will increase. In addition, you often have no control over fonts that are hosted and made available on another server.

Different File Formats for Web Fonts

The different file formats for the fonts seem a bit exotic at first. Not everyone is familiar with font file abbreviations such as *EOT* (*Embedded Open Type*; `format('eot')`), *WOFF* (*Web Open Font Format*; `format('woff')`), *TTF* (*TrueType*; `format('truetype')`), *OTF* (*Open Type*; `format('opentype')`), and *SVG* (*SVG Fonts*; `format('svg')`).

Meanwhile, the WOFF format standardized by the W3C seems to be gaining more and more acceptance. WOFF is a compressed TIFF format with additional information such as the origin or license of the font and is supported by the latest web browsers. The oldest of the formats, EOT, on the other hand, was used by the older Internet Explorer up to version 8. Other older web browsers, on the other hand, used the TTF or OTF formats.

The SVG format is popular for displaying on the iPhone or iPad, but it's also used by Safari. However, you can do without SVG now because iPhone and iPad also support the WOFF format. To provide the widest possible support for the downloadable font, you can provide the fonts in multiple formats.

In practice, this is how you reach almost all web browsers to provide a font in a particular file format:

```
@font-face {
   font-family: 'A font name';
   src: url('path/to/font.eot'); /*IE9*/
   src: url('path/to/font.eot#iefix') format('eot'), /*IE5-8*/
        url('path/to/font.woff') format('woff'),
        url('path/to/font.ttf') format('truetype'),
        url('path/to/font.svg#svgFN') format('svg');
}
```

This way, you can include different formats for different web browsers. If a web browser doesn't support a certain format, it chooses the next possible font format. You could do without TrueType and SVG altogether in this example nowadays. The first EOT file is used for the old Internet Explorer 9. The hash (`#`), in turn, is a browser switch for even older Internet Explorer versions prior to version 9. At this point, the web browser in question stops reading. All other web browsers, however, read their preferred font format.

Having Fonts Converted to Different Formats

When you've found a font that you want to use, you often don't have all the necessary formats such as EOT, WOFF, TTF, or SVG ready. For this purpose, the Font Squirrel website (*www.fontsquirrel.com/tools/webfont-generator*) offers a free service: you can upload a font file and have it converted to all other formats. Then you download the different formats, and the CSS rule with `@font-face` is included in an extra CSS file. However, after embedding web fonts, you should always test the rendering quality in different web browsers because the quality can differ significantly between web browsers when rendering.

In addition, you can use the `font-style`, `font-weight`, and `font-stretch` features in the `@font-face` rule, which is very useful when the font exists in different files, for example:

```
@font-face {
 font-family: 'A font name';
 src: url('path/to/my/font.eot');
 src: url('path/to/my/font.eot#iefix') format('eot'),
      url('path/to/my/font.woff') format('woff'),
      url('path/to/my/font.ttf') format('truetype'),
      url('path/to/my/font.svg#svgFN') format('svg');
}

@font-face {
 font-family: 'A font name';
 src: url('path/to/my/font-it.eot');
 src: url('path/to/my/font-it.eot#iefix') format('eot'),
      url('path/to/my/font-it.woff') format('woff'),
      url('path/to/my/font-it.ttf') format('truetype'),
      url('path/to/my/font-it.svg#svgFN') format('svg');
   font-style: italic;
}
```

This allows you to use the italic font for styling purposes in addition to the regular version of the font, for example:

```
...
p { font-family: 'A font name', Times, Georgia, serif; }
.it { font-style: italic; }
...
<p>Regular version of the embedded font</p>
<p class="it">Cursive version of the embedded font</p>.
...
```

Embedding Royalty-Free Fonts by Google into the Website

The easiest way might be to use a font from Google Fonts and embed it into the website. However, "easy" doesn't mean it's "not complicated" but rather refers to ease of getting the necessary licenses. If you use fonts from the web for a custom project, you really need to be sure that you have the permission to do so from the font's developer. Even a "free" font doesn't always mean that it's free for all purposes. You can find the fonts from Google Fonts listed at *https://fonts.google.com*.

On the upper-left side of the Google Fonts website, you'll see a magnifying glass icon to search with some filters to find the font you need. For the category (**Categories**), you select the type of font (**Serif**, **Sans Serif**, **Display**, **Handwriting**, or **Monospace**). You can filter out other properties using **Font properties** with the thickness or font width. Then, you can select the desired fonts via the preview.

Once you've clicked on a font, you can specify the **styles**. Useful styles are regular, italic, and bold. Click **+ Select this style** to add a font style. The selected styles will be displayed in the **Review** tab on the right.

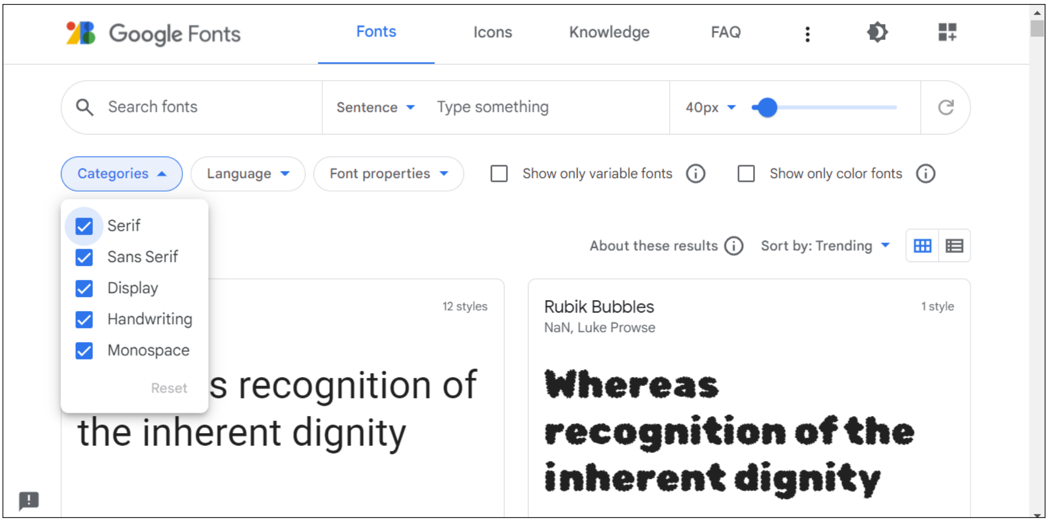
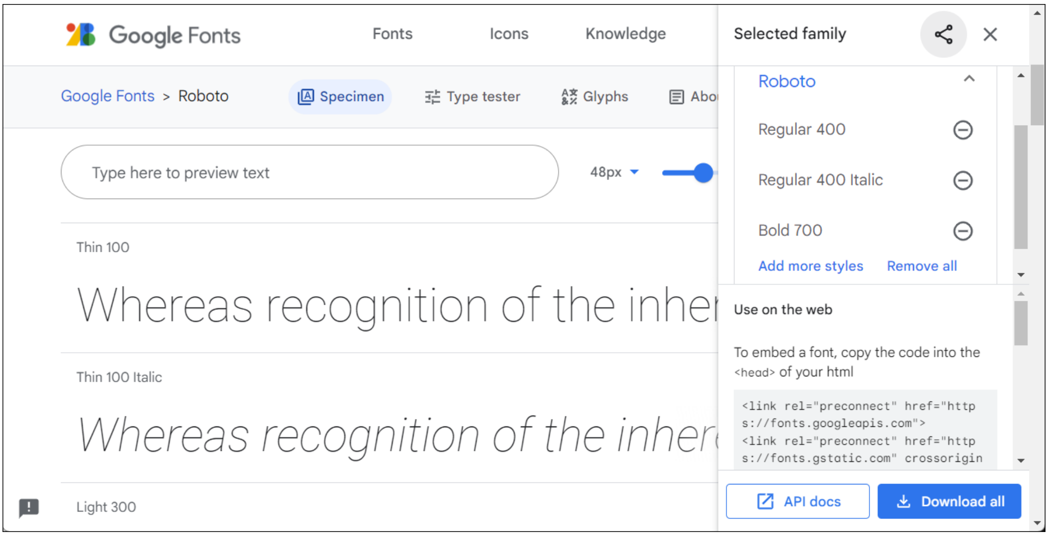

Google Fonts
Fonts
Icons
Knowledge
FAQ
Search fonts
Sentence
Type something
40px
Categories
Language
Font properties
☐ Show only variable fonts
☐ Show only color fonts
☑ Serif
☑ Sans Serif
☑ Display
☑ Handwriting
☑ Monospace
Reset
About these results
Sort by: Trending
12 styles
s recognition of the inherent dignity
Rubik Bubbles
NaN, Luke Prowse
1 style
Whereas recognition of the inherent dignity

Figure 14.6 Fonts on https://fonts.google.com

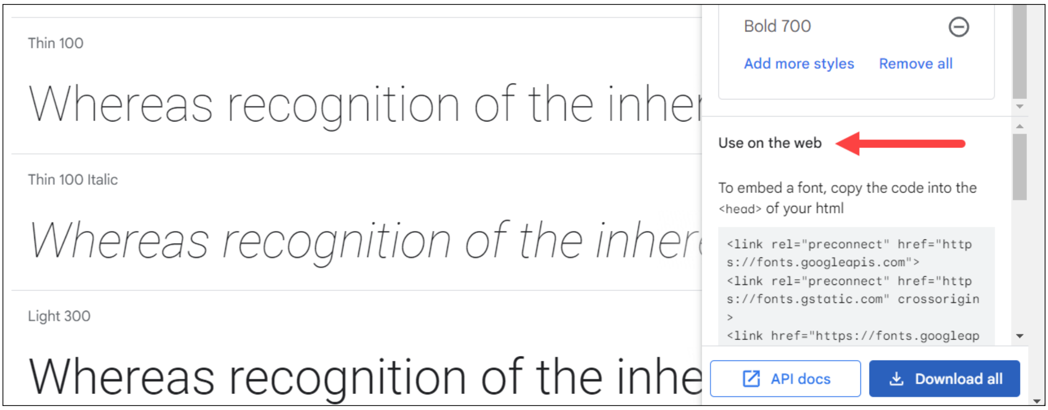

Google Fonts
Fonts
Icons
Knowledge
Google Fonts > Roboto
Specimen
Type tester
Glyphs
Abo
Type here to preview text
48px
Thin 100
Whereas recognition of the inher
Thin 100 Italic
Whereas recognition of the inher
Light 300
Selected family
Roboto
Regular 400
Regular 400 Italic
Bold 700
Add more styles
Remove all
Use on the web
API docs
Download all

To embed a font, copy the code into the <head> of your html

```
<link rel="preconnect" href="http
s://fonts.googleapis.com">
<link rel="preconnect" href="http
s://fonts.gstatic.com" crossorigin
```

Figure 14.7 I've Chosen the Roboto Font with a Regular, Italic, and Bold Font Style

Once you're done with the selection, take a look at the **Review** tab on the right side where you'll find the code to embed the font on your website under the text **Use on the web**. You can choose between a `link` element, an `@import` rule, or a JavaScript. Copy and paste the code to your website.



Thin 100

Whereas recognition of the inher

Thin 100 Italic

Whereas recognition of the inher

Light 300

Whereas recognition of the inhe

Bold 700

Use on the web

To embed a font, copy the code into the `<head>` of your html

```
<link rel="preconnect" href="http
s://fonts.googleapis.com">
<link rel="preconnect" href="http
s://fonts.gstatic.com" crossorigin
>
<link href="https://fonts.googleap
```

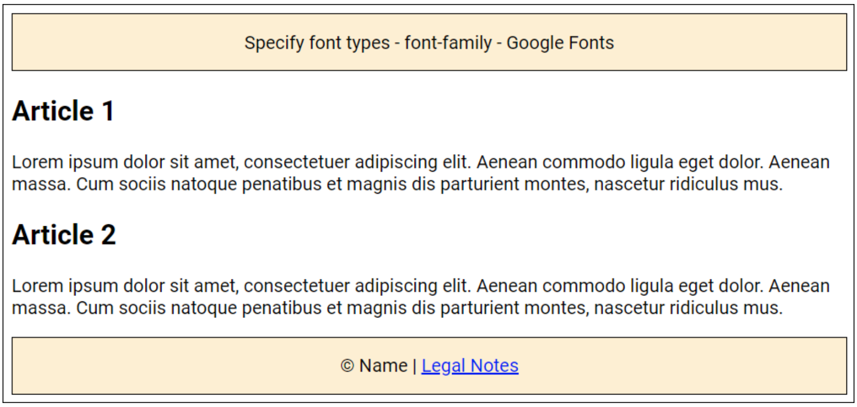
Figure 14.8 A <link> Element or an “@import” Statement Enables You to Add the Code to the Website via Copy and Paste

In the following example, the Roboto font from Google Fonts was embedded and used via `@import`:

```
@import url('https://fonts.googleapis.com/css2?family=
Roboto:ital,wght@0,400;0,700;1,400&display=swap');
body {
    font-family: 'Roboto', sans-serif;
}
...
```

Listing 14.2 /examples/chapter014/14_1_2/css/style.css

You can see the result with the HTML document */examples/chapter014/14_1_2/index.html* in Figure 14.9.



Specify font types - font-family - Google Fonts

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

© Name | Legal Notes

Figure 14.9 Here, the Roboto Font from Google Fonts Was Downloaded and Embedded

Google Fonts and the GDPR

If you offer fonts from a Google server on your website, you may feel insecure about the General Data Protection Regulation (GDPR) because data gets transferred between your website visitors and Google. If you want to be on the safe side, you can also offer Google Fonts "locally" in your own web space. For this purpose, you'll find the google-webfonts-helper tool at *https://gwfh.mranftl.com/fonts* to pack up and download the appropriate font formats.

Other Royalty-Free and Commercial Web Font Providers

Not everyone likes Google's fonts, and if you're looking for a special font, there are other font hosting services as well. Many of these services offer free fonts for private use or even entirely royalty-free fonts. Others offer a mix of free and commercial fonts, and then there are fee-only services. Here's a short list of different hosting services for fonts:

- *www.fontsquirrel.com*
- *http://fontlibrary.org*
- *http://fontsforweb.com*
- *www.fontspring.com*
- *https://fonts.adobe.com*

Note that in the services where you can download the corresponding fonts, you usually receive a text file with the license in addition to the font file. Be sure to read them so you know under what conditions you may use the fonts. If you buy a font, you should make sure you have the license to use it as a web font (not just for the desktop).

Pros and Cons of "@font-face"

The advantage of `@font-face` is certainly that it allows you to finally use fonts that aren't installed on the computers of your website visitors. However, this font needs to be downloaded beforehand, which might slow down the loading of the website a bit. In addition, here you need to know the license of the font used and whether the distribution of the font is allowed or not. If you want to be absolutely sure in this regard, you need to either create your own fonts or use `@font-face` via a free or commercial service. In that case, the service takes care of the licensing arrangements with the font manufacturer.

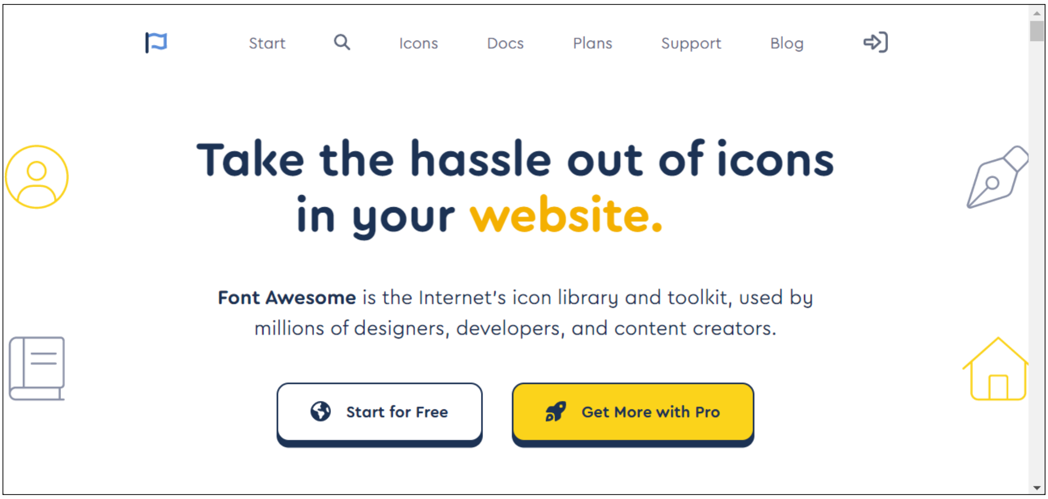
14.1.3 Using Icons via Icon Fonts

Adding graphics to the website is basically no big deal anymore. However, it gets somewhat more complicated if you want to insert an icon in the middle of a text. And that's particularly difficult when you want the icon to look equally good on any device, from a small screen such as a smartphone to a screen with an extremely high resolution. Sure, you could make the icon responsive as a graphic and scale it accordingly, but it won't necessarily make the result more attractive (blurry or pixelated). On the other hand, you could also provide multiple versions of the graphic, and SVG as a graphic format still comes to mind as a possible workaround.

However, you can save this effort right away and just use *icon fonts* instead. The icon fonts name already suggests what it's about, and those who have a lot to do with word processing might know fonts such as *Wingdings* from Microsoft, which use icons. You only need to include the corresponding icon fonts via `@font-face`. This makes it possible to treat these icons like an ordinary font. For example, you can adjust the appropriate size with the CSS feature `font-size`.

There are several providers of attractive icon fonts. Here we'll introduce and use one of the arguably more popular icon fonts, *Font Awesome* (*https://fontawesome.com*). A list of other popular icon fonts can be found at the end of the section. To download icon fonts, you need to create an account at fontawesome.com. However, it is also possible to use the icon fonts via a CDN server. For more details, please refer to the fontawesome.com website.

If you've downloaded and unpacked Font Awesome, you'll also find other embedding options in the package, such as for the CSS preprocessor *Less*, for example. For our purposes, the contents of the *css* and *webfonts* folders, which are located below the *webfonts-with-css* folder, are sufficient for now. The *css* folder contains all the necessary CSS statements, while you can find the font icons in *webfonts*. Both folders were copied to the directory of the sample website.



Take the hassle out of icons in your website.

Font Awesome is the Internet's icon library and toolkit, used by millions of designers, developers, and content creators.

Figure 14.10 Font Awesome Has Become a Favorite of Web Designers

In the first step, you don't need to do anything but include the CSS file *all.css* in the HTML document, which you can do using the `link` element in the document header as follows:

```
...
<link href="css/all.css" rel="stylesheet">
...
```

That's all it takes. The included CSS file from Font Awesome handles the integration of the icon fonts with `@font-face` as an additional font. The second step is to use Font Awesome's font icons anywhere in the HTML document using the `<i>` tag, but basically this works with any other tag as well. For example, you can insert and display a house icon in the HTML document as follows:

```
<i class="fas fa-home"></i>
```

You can adjust the size with `font-size` as you would with an ordinary font because basically they are embedded font icons, for example:

```
<i class="fas fa-home" style="font-size:3em;"></i>
```

The font icons of Font Awesome have special classes included that allow you to increase the icon size with `fa-2x`, `fa-3x`, `fa-4x`, and `fa-5x` relative to their container, for example:

```
<i  class="fas fa-home fa-2x"></i>
```

You can also customize colors as you would with an ordinary font using the CSS feature `color`. For example, the following line uses the Twitter logo in Twitter's usual color:

```
<i class="fab fa-twitter fa-2x" style="color:#0084b4;"></i>
```

For symbols with a trade name or trademark, you must use the `fab` prefix instead of `fas`. The `b` of `fab` stands for *brand*, and the `s` of `fas` for *solid*. Font Awesome also offers a commercial version with even more icons and different styles, where you can use `far` (for *regular*) and `fal` (for *light*) as prefixes.

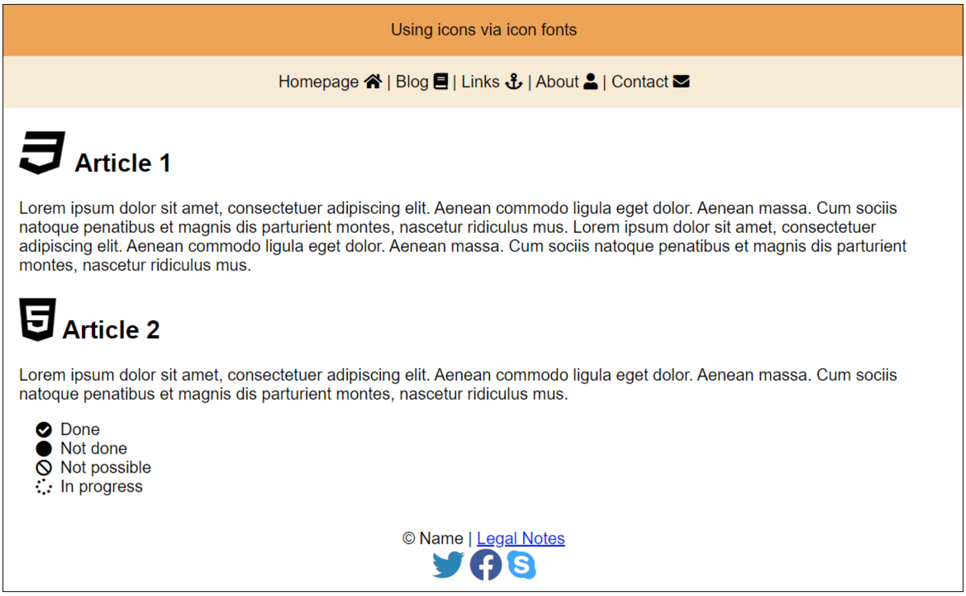
For an overview of Font Awesome's icons, you can visit *https://fontawesome.com/icons?d=gallery*. To find out what else you can do with Font Awesome, go to *https://fontawesome.com/start*.

Here is a snippet of an HTML document with various font symbols from Font Awesome in use:

```
...
  <head>
...
    <link href="css/all.css" rel="stylesheet">
    <style>
    ...
    <style>
  </head>
  <body>
    <header class="header">Use icon font</header>
    <nav>
    <a href="#">Home page <i class="fas fa-home"></i></a>|
    <a href="#">Blog <i class="fas fa-book"></i></a> |
    <a href="#">Links <i class="fas fa-anchor"></i></a> |
    <a href="#">About <i class="fas fa-user"></i> </a> |
    <a href="#">Contact <i class="fas fa-envelope"></i>
    </a>
    </nav>
    <main>
      <article>
        <h1><i class="fab fa-css3 fa-2x"></i> Article 1</h1>
        <p>Lorem ipsum dolor sit ... </p>
      </article>
      <article>
        <h1><i class="fas fa-html5 fa-2x"></i> Article 2</h1>
        <p>Lorem ipsum dolor sit ... </p>
        <ul class="fa-ul">
          <li><i class="fa-li fas fa-check-circle"></i>
              Done</li>
          <li><i class="fa-li fas fa-circle"></i>
               Not done</li>
          <li><i class="fa-li fas fa-ban"></i>
              Not possible</li>
          <li><i class="fa-li fas fa-spinner fa-spin"></i>
              In process</li>
        </ul>
      </article>
    </main>
    <footer class="footer">&copy; Name | <a href="#">Legal Notes</a><br><br>
    <i class="fab fa-twitter fa-2x" style="color:#0084b4;"></i>
    <i class="fab fa-google-plus fa-2x" style="color:red;"></i>
    <i class="fab fa-facebook fa-2x" style="color:#3b5998;"></i>
    <i class="fab fa-skype fa-2x" style="color:#12A5F4;"></i>
    </footer>
  </body>
...
```

Listing 14.3 /examples/chapter014/14_1_3/index.html

You can see the example with the different icon fonts, including social media icons, from Font Awesome in Figure 14.11.



Using icons via icon fonts

Homepage 🏠 | Blog | Links ⚓ | About | Contact ✉

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

- ✓ Done
- ● Not done
- ⃠ Not possible
- ⁘ In progress

© Name | Legal Notes

Figure 14.11 Various Icons without Graphics in Use Thanks to Font Awesome Icon Fonts

Besides Font Awesome, there are many other providers of such font icons, which can often be integrated and used in a similar way. Other interesting icon fonts can be found on the following websites, among others:

- *http://genericons.com*
- *http://icomoon.io*
- *http://fontello.com*
- *http://glyphicons.com*
- *http://www.entypo.com*

Observing Licenses

The same applies here as with the downloadable fonts. Many of these icon fonts are free, but still have some sort of license (GPL, Creative Commons, etc.) that you should be sure to read through before using and embedding icon fonts on your website. Others are commercial and can be purchased.

14.1.4 Setting the Font Size Using “font-size”

The font size can be set using the CSS feature `font-size`. As a matter of fact, you might think it should be trivial to set the font size. But it already starts with the fact that the font size can be specified with pixels, points, percentages, `em`, or `rem`. The ideal font size will probably not exist anyway because there are too many different settings in the operating system and different large and small screens with different resolutions. In addition, the web browser allows you to scale the websites in different zoom levels.

Different screen sizes and resolutions, settings in the operating system or web browser, and different units of measure (UoM) make it truly complicated for the web designer to use the right font size and UoM. Nevertheless, in this chapter, you'll learn what you can use when and what you should not use.

No Specifications with "font-size": The Browser Standard

If you don't specify anything via `font-size`, the default value of the web browser will be used, which is often 16 pixels (= 100%, `1em`, `1rem`, or `16pt`) as the base font size. Because default fonts are often displayed at different sizes, and users can change the size in the web browser, you should take control of the font size and not leave the display of text to chance.

Preset Keywords for the Font Size

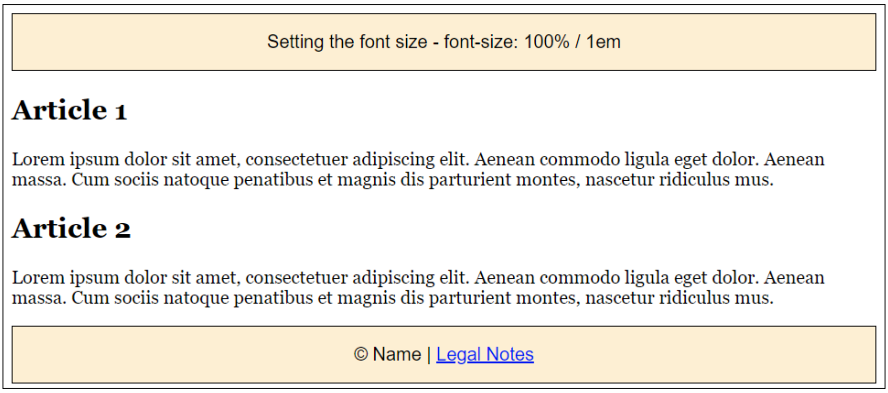
CSS provides the predefined keywords `small`, `x-small`, `xx-small`, `medium`, `large`, `x-large`, and `xx-large`, where `medium` is the base font size. The other keywords decrease (`small`) or increase (`large`) the value of `medium` by a factor of 1.2 each. These values are absolute values. `smaller` and `larger` are two more keywords with relative values. Relative here means relative to the parent element. I personally have rarely made use of these keywords, as they allow only limited control over the actual font size. For this reason, I won't go into detail about those values.

Relative Font Sizes with "em"

An easy way to adjust the font size for the entire document is to set `font-size` for the `body` element. For example, if you set `font-size` for the `body` element to `1em` or `100%`, you'd have effectively used the default value of the web browser, which is the case in Figure 14.12.

If you want to increase the font size by 15% for the complete document, you only need to set `font-size` in the `body` element to `1.15em` or `115%`. This will automatically increase the font size of the other elements such as `<h1>` and `<p>` by 15%, and you don't have to worry about that. In Figure 14.13, compared to Figure 14.12, this is exactly what was done: the font was increased by 15% via the `body` element.

Due to the fact that a relative font size of the `body` element regulates the font size for all elements of the web page through inheritance, this option is widely used in practice to adjust the font size.



Setting the font size - font-size: 100% / 1em

Article 1

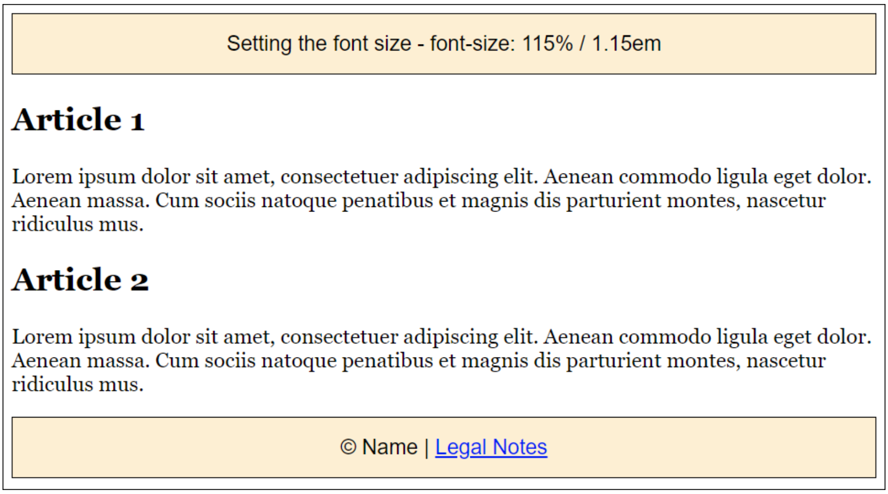
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

© Name | Legal Notes

Figure 14.12 The Default Font Size Gets Preserved If You Set "font-size" to 100% or "1em" for the <body> Element



Setting the font size - font-size: 115% / 1.15em

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

© Name | Legal Notes

Figure 14.13 Here, the Font Size Has Been Increased by 15% via the <body> Element

But it's precisely this inheritance or, more accurately, cascading, that can make adjusting font sizes a little more complex if you don't act with caution here. Consider the following theoretical example:

```
...
  body    { font-size: 0.95em;  /* or 95% */ }
  article { font-size: 0.8em;   /* or 80% */ }
  p       { font-size: 0.8em;   /* or 80% */ }

...
```

Listing 14.4 /examples/chapter014/14_1_4/css/style.css

```
...
<body> <!-- 0.95em -->
...
  <article> <!-- 0.76em -->
    <h1> ... </h1>
    <p> <!-- 0.608em --> ... </p>
  </article>
...
</body>
...
```

Listing 14.5 /examples/chapter014/14_1_4/index.html

For `<body>`, we set a smaller font size of `0.95em` (or 95%). As mentioned earlier, this specification applies to the entire document, which has the side effect of reducing the `article` element by another `0.8em` (or 80%) of this `0.95em` (or 95%), thus setting it to `0.76em`, since 0.96 × 0.8 = `0.76em`.

In the example, the `p` element is still used inside an `article` element, which again reduces the set font size of `0.76em` by `0.8em`, so that a text inside the `p` element in an `article element` is set only in the font size `0.608em` (0.76 × 0.8 = 0.608). The text in the `p` element would thus be displayed extremely small.

Setting the Font Size Using "rem"

The problem with inheriting relative values that occur if you use `em` or `%` for setting the font size can be avoided by using `rem` (`rem` = *root em*). Basically, `rem` is also just an `em`, the only difference being that when it inherits, it adheres to the highest root element, `<html>`, instead of the font size of the corresponding parent element. Let's take a look at the same section as before, but this time we use `rem`:

```
...
  html    { font-size: 100%; } /* Browser default */
  body    { font-size: 0.9375rem; }
  article { font-size: 0.8125rem; }
  p       { font-size: 0.8125rem; }
...
```

Listing 14.6 /examples/chapter014/14_1_4/css/style2.css

```
...
<body> <!-- 0.9375em -->
...
  <article> <!-- 0.8125em -->
    <h1> ... </h1>
    <p> <!-- 0.8125em --> ... </p>
  </article>
...
</body>
...
```

Listing 14.7 /examples/chapter014/14_1_4/index2.html

Here, you don't need any more math like in the example before with `em` because we've set the value of `<html>` to 100%, and on the basis of `<html>`, the *root em*, you can be sure that the subsequent font sizes with `rem` correspond to what has been written. Of course, the relationship of body text such as `<p>` to headings such as `<h1>`, `<h2>`, and so on will be preserved.

Fixed Defaults for the Font Size via Pixel and Point

For a long time, font sizes were specified in pixels. People were familiar with this UoM from the screen, and they could avoid the problem with the inheritance of relative values, as it was the case with percentages or with `em`. In addition, pixels can be used to implement a pixel-precise layout.

However, 12 pixels looks different on a 1,024 × 768 pixel 13-inch screen than on a 27-inch screen with a resolution of 2,560 × 1,440 pixels. In addition, it's no longer possible to say that a pixel is just a pixel. The days of uniform pixel densities of 72 ppi or 96 ppi are over. Newer devices such as smartphones often have 326 ppi, or various screens like the Retina of the iMac have a pixel density of 104 ppi. Without going too much into the complex details, this specifically means that with a higher pixel density on an inch, the pixels inevitably become smaller (and therefore the resolution becomes sharper). We hardly notice the individual pixels on a smartphone due to the high pixel density.

What about Points ("pt") as a Unit for Font Size?

The point value (`pt`) is better suited for printers or typesetters in the print sector. However, the different conversion factors of the pixel densities from 72 ppi up to 326 ppi result in different displays. The `pt` unit is more suitable for printing if you create a print version with CSS. It should be mentioned here that a specification of `cm` (for centimeters) is also possible. As with `pt`, it's also true for `cm` that it isn't possible to say whether the conversion to pixels has been performed correctly, and therefore the results of these specifications are relatively unpredictable.

For this reason, if a smartphone actually used 12 pixels for a font size at this high pixel density, you would have to use a magnifying glass to find the text. As a result, such mobile web browsers convert the device's pixels into a kind of pixel for CSS. Consequently, 12 pixels aren't always really 12 pixels. Thus, specifying pixels is rather unreliable with the extremely different sizes and resolutions of screens that exist today.

Responsive Units “vw” and “vh”

The font sizes with `%`, `em`, and especially `rem` are probably the most common units at the moment. These specifications are relative to the base font size or relative to the parent element. What’s still missing here is a font size specification, which is relative to the screen dimensions. For this purpose, the W3C has introduced the *viewport units*, `vw` (for *view width*) and `vh` (for *view height*), which allow you to assign a size to an element that’s calculated in relation to the width and height of the viewport. For the width, you can use `vw`, and `1vw` corresponds to 1% of the width of the viewport. Similarly, the same applies to the height where you can use `vh`, and `1vh` corresponds to 1% of the height of the viewport. In addition, the units `vmin` and `vmax` are available, which refer to the height or width, using the smaller or larger value, respectively. Again, `1vmax` corresponds to 1% of the width or height of the viewport.

Here’s a simple example of how you can adjust the font size without media queries based on the screen width with just a single specification:

```
html { font-size: 3vw; }
...
```

Listing 14.8 /examples/chapter014/14_1_4/css/style3.css

If you now run the example */examples/chapter014/14_1_4/index3.html* on different devices or scale the browser window, the font size will always be scaled by `3vw` according to the screen width. This can be converted as follows if, for example, the screen width is 1,024 pixels:

```
1024px / 100 * 3vw = 30.72 pixels
```

This would have set the general font size of the web browser to 30.72 pixels for a screen width of 1,024 pixels. On a smaller screen width with a 480-pixel viewport this would be as follows:

```
480px / 100 * 3vw = 14.4 pixels
```

The example shows very nicely how you can achieve extremely responsive font specifications with the viewport unit, but in the example with `3vw`, the font size in the `html` element is now much too large on large screens and barely legible on smaller screens.

Mike Riethmuller has found a solution to the problem (see *www.madebymike.com.au/writing/precise-control-responsive-typography/*), where he limits the scaling of the font size using `calc()`:

```
html { font-size: calc(100% + 0.5vw); }
...
```

Listing 14.9 /examples/chapter014/14_1_4/css/style4.css

When you run the example with */examples/chapter014/14_1_4/index4.html*, you'll notice that everything is far from perfect, and the question remains how to specifically adjust individual elements with the viewport units. Just giving the `vw` or `vh` information seems to be too inaccurate. Further calculations with `calc()` might make everything a little too complicated. Probably the best solution is to set the viewport units only in the `html` element and use `em` or `rem` for everything else relative to it, as is also currently recommended by Zell Liew at *https://zellwk.com/blog/viewport-based-typography/*.

This was just intended as a brief introduction to the newer viewport units related to font sizes. We'll probably encounter the new unit more often in the future.

General Relative Length Measure

The units `vw` and `vh` aren't limited to fonts, but were introduced as a general length measure, which you can already conclude from the names *viewport width* and *viewport height*. As mentioned previously, `1vw` corresponds to 1% width of the web browser window. Thus, `100vw` is the full browser width. This makes it easy, for example, to make sure that an element is always a certain size, no matter how big the screen is. Here's an example:

```
.quarter {
  width: 50vw;
  height: 50vh;
}
```

The element with the `.quarter` class will now always cover a quarter of the browser window, no matter how large the device's screen is or whether you scale the browser window afterward.

Overview of the Common Methods for "font-size"

Table 14.2 provides a brief overview of the common methods or units you can apply to the CSS feature `font-size`.

Unit	Example	Description
`em`	`font-size: 1em;`	Relative to the font size of the parent element
`%`	`font-size: 100%;`	Relative to the font size of the parent element
`px`	`font-size: 16px;`	Absolute font size
`rem`	`font-size: 1rem;`	Relative to the font size of `<html>`
`smaller`, `larger`	`font-size: larger;`	Slightly larger than the parent element

Table 14.2 Common Ways to Set the Font Size

Unit	Example	Description
`small-x`, `medium`, ...	`font-size: small-x;`	Uses exactly `small-x` (absolute font size)
`vw`	`font-size: 3vw;`	Adjusts the font size according to the width of the viewport (see next section)

Table 14.2 Common Ways to Set the Font Size (Cont.)

Converting Pixels to "em" or "rem" with the 62.5% Trick

Richard Rutter's 62.5% trick is frequently encountered in responsive web design. Because most web browsers set the default font size to 16 pixels, and, based on that, 1 rem (or 1 em) always corresponds to the base font size, it makes sense to set the base font size to 10 pixels so that you can more easily set the relative values via `em` or `rem`. For example, if you want to set the text for an element to 18 pixels, you'd have to write a cumbersome specification such as `1.125em`. You can find a table on this at *http://pxtoem.com*. Of course, you can also calculate this with 18 px ÷ 16 px = 1.125 em, but this is cumbersome in the long run. One option would be to use `1.8em` right away for an 18-pixel font size. To do that, you just need to write the following definition:

```
body { font-size: 62.5%; /* base font size to 10 pixels */ }
h1 { font-size: 2.4em; /* = 24 pixels */ }
h2 { font-size: 1.9em; /* = 19 pixels */ }
...
```

14.1.5 Italic and Bold Fonts via "font-style" and "font-weight"

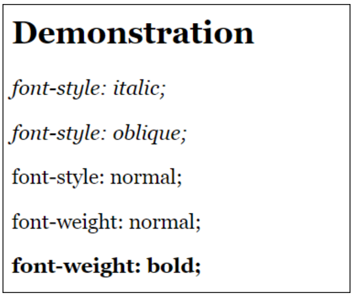
You can assign an italic font style by assigning the value `italic` to the CSS feature `font-style`. This will display the font in italic style. If a font doesn't have an `italic` style, the web browser will try to slant it using `oblique`, which is another value you can assign to `font-style`. The difference between `italic` and `oblique` isn't apparent at first glance, but `italic` uses real italics supplied by the font's manufacturer. With `oblique`, on the other hand, you can subsequently slant the fonts so that they look like real italic fonts. The default value of the CSS feature `font-style` is `normal`, with which the font gets displayed normally upright.

The CSS feature `font-weight` enables you to define the *weight* of the font. The term weight describes how thick or bold the letters will be displayed. The `bold` value allows you to define a bold font style. The default value of the normal font style is `normal`. In addition to `bold`, there are other weights such as `bolder` (bolder than `bold`), `lighter` (thinner than `normal`), and the numeric values `100`, `200`, and up to `900` (in increments of 100), where `400` is `normal` and `700` is `bold`. The question as to how strongly the font will be displayed with the values `100` to `900` depends on the computer platform and the web browser.

For Me, Only “bold” and “normal” with “font-weight” Works!

Most of the time, the web browser only recognizes the `bold` and `normal` font styles. Values such as `lighter`, `bolder`, or `100` to `900` can only be used if the font has these gradations.

In Figure 14.14, you can see a trivial example that demonstrates the CSS features `font-style` and `font-weight` for adjusting the font style with CSS.



Demonstration

font-style: italic;

font-style: oblique;

font-style: normal;

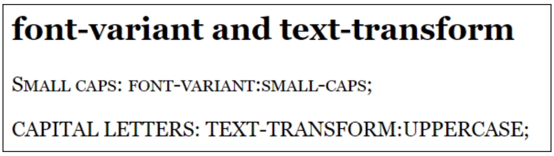
font-weight: normal;

font-weight: bold;

Figure 14.14 Changing the Font Style with “font-style” and “font-weight” (Example in /examples/chapter014/14_1_5/index.html)

14.1.6 Creating Small Caps Using “font-variant”

With `font-variant` and the single value `small-caps`, you can turn a letter into small caps. By means of a small cap, the text is converted to all uppercase, while maintaining the size of the lowercase letters. As a rule, true small caps, in which all letters have the same stroke width, aren’t used unless the font used contains small caps. With the help of `@font-face` and an appropriate font, it’s possible to use real small caps.



font-variant and text-transform

SMALL CAPS: FONT-VARIANT:SMALL-CAPS;

CAPITAL LETTERS: TEXT-TRANSFORM:UPPERCASE;

Figure 14.15 The Difference between (Fake) Small Caps and Capital Letters (Example in /examples/chapter014/14_1_6/index.html)

Capital Letters

If you want to convert a text to uppercase, you can do this by using the CSS feature `text-transform` and the `uppercase` value (Section 14.1.14).

14.1.7 Defining Line Spacing via “line-height”

The line spacing defines the distance from baseline to baseline and can be set using the CSS feature `line-height`. Line spacing is important for better readability of longer text passages. In practice, the default value on the monitor is almost always too small because this distance comes from the print area. For this reason, you’re well advised to use a higher value. Most of the time, the following applies: the longer the lines of a text are, the larger you should choose the line spacing. A good value is often 120% (or `1.2em`) up to 150% (or `1.5em`). An increased line spacing helps your visitor “keep” the line while reading.

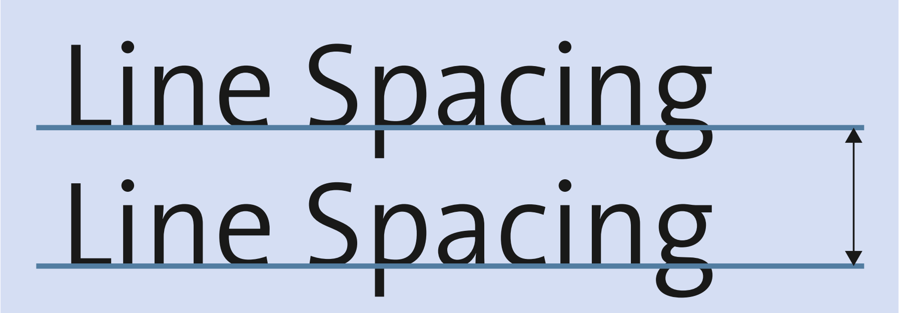


Figure 14.16 Line Spacing Is the Distance from Baseline to Baseline

Line spacing is often confused with the optical bleed-through shown in Figure 14.17.

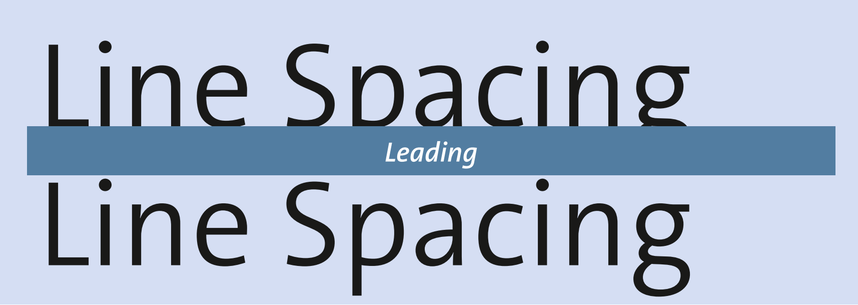
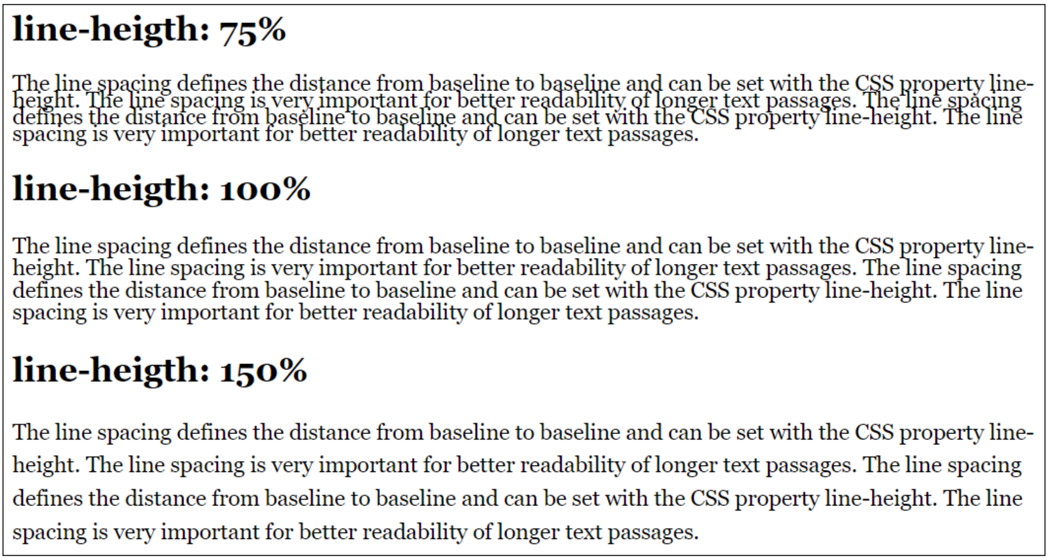


Figure 14.17 Don’t Confuse the Optical Bleed-Through with Line Spacing

In Figure 14.18, you can see how different values for the CSS feature `line-height` have a significant impact on the readability of the body text. You can also clearly see that a value below 100% reduces the line spacing and drastically worsens the readability of the continuous text because you can no longer keep the line as easily when reading.



line-heigth: 75%

The line spacing defines the distance from baseline to baseline and can be set with the CSS property line-height. The line spacing is very important for better readability of longer text passages. The line spacing defines the distance from baseline to baseline and can be set with the CSS property line-height. The line spacing is very important for better readability of longer text passages.

line-heigth: 100%

The line spacing defines the distance from baseline to baseline and can be set with the CSS property line-height. The line spacing is very important for better readability of longer text passages. The line spacing defines the distance from baseline to baseline and can be set with the CSS property line-height. The line spacing is very important for better readability of longer text passages.

line-heigth: 150%

The line spacing defines the distance from baseline to baseline and can be set with the CSS property line-height. The line spacing is very important for better readability of longer text passages. The line spacing defines the distance from baseline to baseline and can be set with the CSS property line-height. The line spacing is very important for better readability of longer text passages.

Figure 14.18 Different Line Spacing Has a Drastic Effect on the Readability of the Text (Example in /examples/chapter014/14_1_7/index.html)

14.1.8 A Short Notation for Font Formatting Using "font"

The CSS feature `font` is a short notation for all the features presented here in the order `font-style`, `font-variant`, `font-weight`, `font-size/line-height`, and `font-family`. In practice, this short CSS notation is preferably placed in the `<body>` tag, and for individual elements such as headings or paragraph text, only the individual adjustments are made for it. For example:

```
body { font: 1.125em/150% Arial, sans-serif; }
footer, header {
  ...
  font-size: 1.2em;
}
h1 { font-style: italic; }
article {
  font-family: Georgia, Times, serif;
  font-size: 1em;
}
...
```

Listing 14.10 /examples/chapter014/14_1_8/css/style.css

Here, the font size was set to `1.125em` for the `<body>` tag, the line height to 150%, and the font to Arial or any existing serif font using the CSS feature `font`. By using this line, you've virtually defined the font for the website. For all other variations, as you can see in the example for the `footer`, `header`, `h1`, and `article` tags, you only need to adjust the individual characteristics for that font.

As you can see with the CSS feature `font`, you don't have to specify all properties. At least the `font-size` and `font-family` features must be present. In addition, if you use `font-size` and `line-height`, you must separate the two with a slash, where the first value is for `font-size`, and the second is for `line-height`. If only one value is used, it will apply to `font-size`. Here's a summary of the sequence that must be followed when using all features:

```
font: font-style                /* font style */
      font-variant              /* font variant */
      font-weight               /* font weight */
      font-size/line-height     /* font size/line spacing */
      font-family;              /* font family */
```

A complete example in which all `font` features have been combined can look like the following:

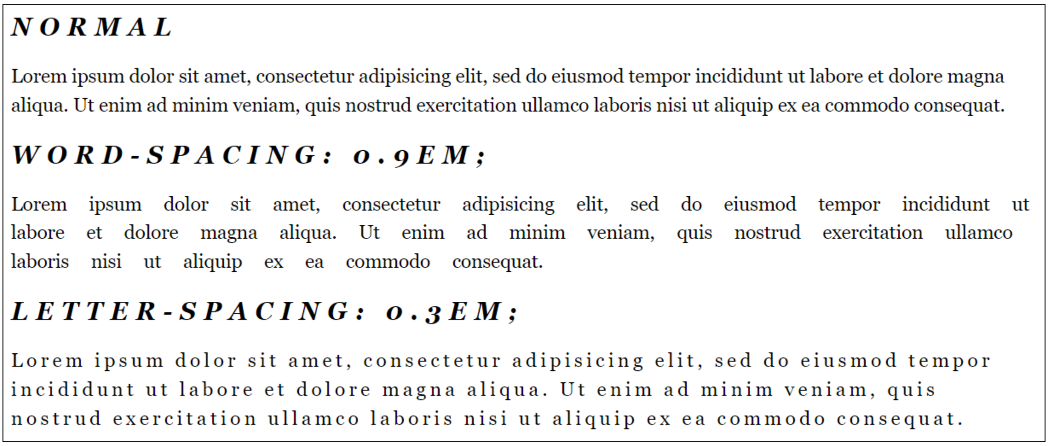
```
p { italic normal bold 1.2em/120% Georgia, Times, serif; }
```

14.1.9 Specifying Letter and Word Spacing via "letter-spacing" and "word-spacing"

If you want to control the spacing between the letters, you can do this by using the CSS feature `letter-spacing`. In your daily work, you shouldn't use letter spacing in regular body text because it tends to make the text less readable. This feature might be more useful for headlines or for texts that are written entirely in capital letters.

What `letter-spacing` does with individual letters, the CSS feature `word-spacing` can do with the spacing between individual words. By default, this kind of spacing is usually `0.25em`, but as always, it depends on the default setting of the web browser. Here, too, a wider specification tends to worsen the reading flow and should therefore only be used sporadically.

In Figure 14.19, you can see the effects of `word-spacing` and `letter-spacing` used after the heading for the corresponding paragraph text. The headlines here were also styled using `letter-spacing`. The example can be found in */examples/chapter014/14_1_9/index.html*.



NORMAL

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

WORD-SPACING: 0.9EM;

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

LETTER-SPACING: 0.3EM;

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Figure 14.19 CSS Features “word-spacing” and “letter-spacing” in Use

Controlling the Width of the Individual Characters via “font-stretch”

The `font-stretch` feature can be used to change the width of the font by compressing or stretching the individual characters. However, this feature can’t be applied to any font, but works only for fonts that contain appropriate subsets for it. Possible values in ascending order of font compression are `semi-condensed`, `condensed`, `extra-condensed`, and `ultra-condensed`. Values that stretch the text, on the other hand, are `semi-expanded`, `expanded`, `extra-expanded`, and `ultra-expanded`. The default setting, which doesn’t change the font width, is `normal`. There are fonts that can handle all nine values. Some fonts, on the other hand, can only be compressed via `condensed` and stretched via `expanded`. When this book went into print, all modern browsers were able to handle this feature, except for Safari for iOS. For an example, see */examples/chapter014/14_1_9/index2.html*.

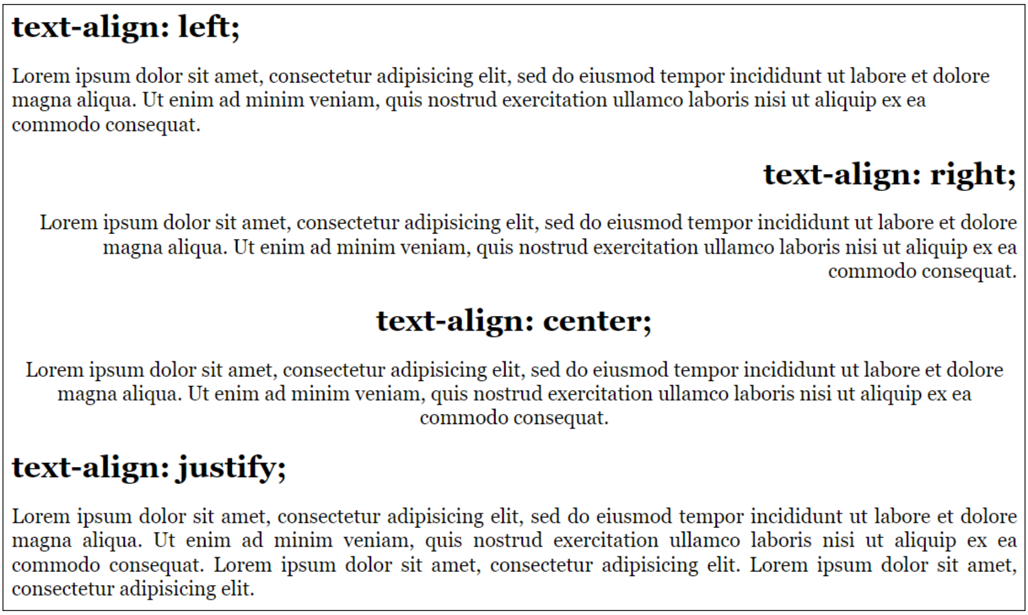
14.1.10 Setting the Text Alignment Using “text-align”

Another important aspect for a good reading flow of texts (and also other inline elements) is the alignment—also referred to as *font type*—which can be set to one of the following four values using the CSS feature `text-align`:

- `left`
  This left-aligns the text, and is usually the default setting of the web browser. In a left-justified text with ragged margins, the line beginnings of all lines are in a perpendicular alignment to each other. This left-justified alignment is most often used on websites because texts can be read best that way.
- `right`
  This aligns the text to the right.

- `center`
  This value is used to align the text centered (also called *axial alignment*). This is a uniform type setting in which the lines of a text are aligned exactly along a central axis. For ordinary and longer paragraph text, the center alignment is less suitable for reading. Centered text, on the other hand, can be useful for headlines, poems, or short texts.
- `justify`
  This aligns the text in justified type, with each line the same width and flush left and right. Justification is mainly used in book and newspaper typography. While justified text can look prettier than left-justified text with ragged margins, it can result in unsightly larger gaps between words because the web browser tries to keep the text flush on the right and left, which disrupts the flow of reading. CSS has introduced a `hyphenation` option via the CSS feature `hyphens`, but currently not all web browsers (see *http://caniuse.com/css-hyphens*) can handle it without any problem.

In Figure 14.20, you can see the different effects of `text-align` on a paragraph text; the best reading flow is achieved with a left-aligned or justified alignment. Because justified text can result in unsightly gaps, left justification is probably still the first choice for websites. For short texts or headlines, a centered alignment can be interesting. The example for this figure can be found in */examples/chapter014/14_1_10/index.html*.



text-align: left;

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

text-align: right;

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

text-align: center;

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

text-align: justify;

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Lorem ipsum dolor sit amet, consectetur adipisicing elit. Lorem ipsum dolor sit amet, consectetur adipisicing elit.

Figure 14.20 Effects of "text-align" on Paragraph Text

14.1.11 Setting the Vertical Alignment via “vertical-align”

The CSS feature `vertical-align` can be used for the vertical alignment of inline elements and isn’t intended for block elements such as `<p>` or `<div>`.

This makes it easy to align text or images in a table cell using a baseline, for example. For table cells, you can use the values `top`, `middle`, or `bottom`. In Figure 14.21, you can see these three values when executed in table cells that have been vertically aligned with the following rows:

```
...
  .vtop { vertical-align: top; }
  .vmiddle { vertical-align: middle; }
  .vbottom { vertical-align: bottom; }
  .vsuper { vertical-align: super; }
  .vsub { vertical-align: sub; }
  .vsub-05em { vertical-align: -0.5em; }
...
```

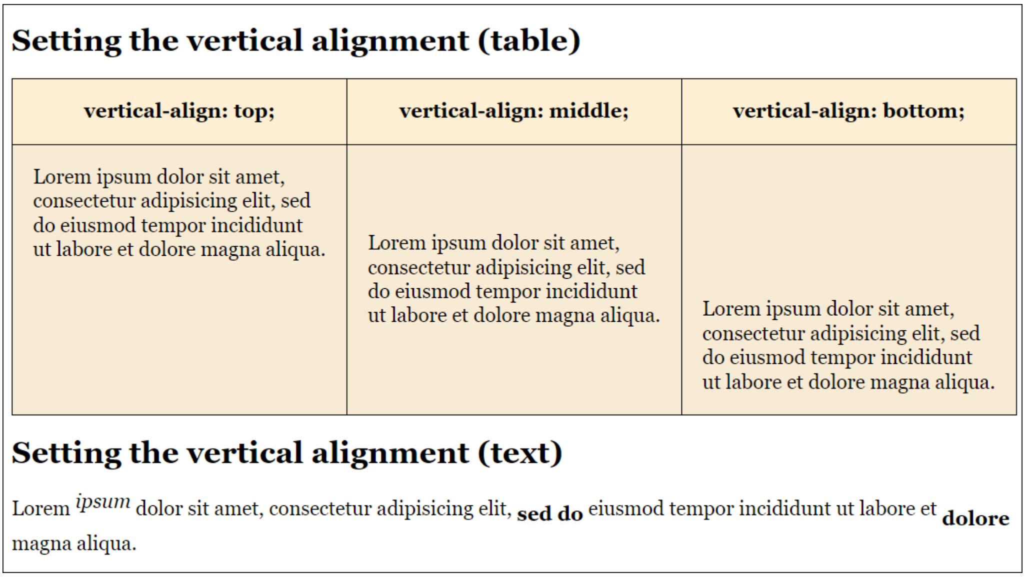
Listing 14.11 /examples/chapter014/14_1_11/css/style.css

Similarly, you can align inline elements in texts based on a baseline relative to the text by using `vertical-align`. Here you align the text with the value `baseline` on the baseline, with `sub` below it and with `super` above the baseline. If aligning above or below the baseline doesn’t suffice for you, you can use positive or negative values of percent, (r)em, or pixels to set the elements even higher or even lower. Here’s an example where the inline elements `<em>` or `<strong>` have been set higher or lower, respectively, via `vertical-align`, which you can see in Figure 14.21:

```
...
<p>Lorem <em class="vsuper">ipsum</em>
   dolor sit amet, consectetur adipisicing elit,
   <strong class="vsub">sed do</strong> eiusmod
   tempor incididunt ut labore et
   <strong class="vsub-05em ">dolore</strong>
   magna aliqua.
</p>
...
```

Listing 14.12 /examples/chapter014/14_1_11/index.html

You can also use the CSS feature `vertical-align` for images with `<img>`. Note that a vertical alignment of the image at the top edge isn’t the same as a `float`. In contrast to `float`, the image with `vertical-align` still occupies one line, as you can see clearly in Figure 14.22.



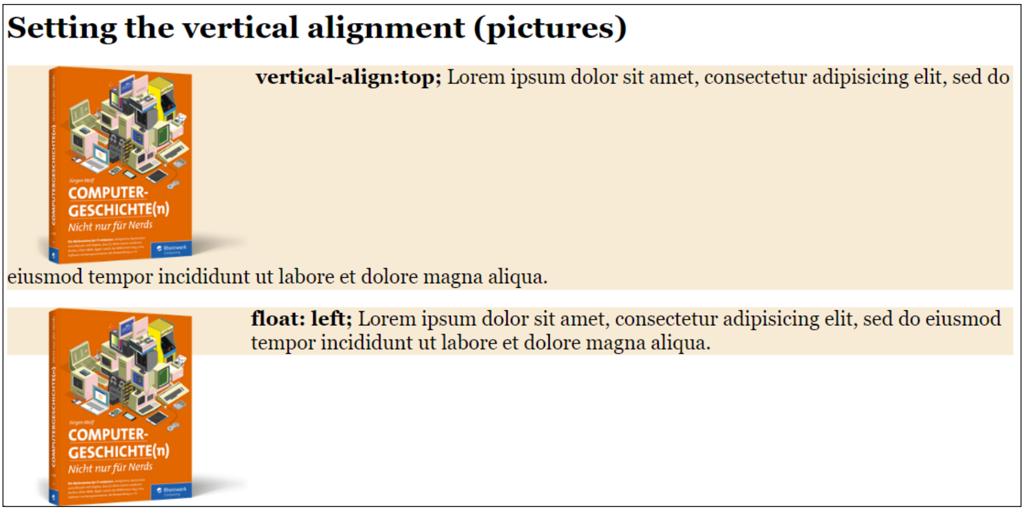
Setting the vertical alignment (table)

vertical-align: top;	vertical-align: middle;	vertical-align: bottom;
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.	Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.	Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Setting the vertical alignment (text)

Lorem *ipsum* dolor sit amet, consectetur adipisicing elit, **sed do** eiusmod tempor incididunt ut labore et **dolore** magna aliqua.

Figure 14.21 Vertical Alignment of Text in Table Cells and of Inline Elements in Text on the Baseline



Setting the vertical alignment (pictures)

vertical-align:top; Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

float: left; Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

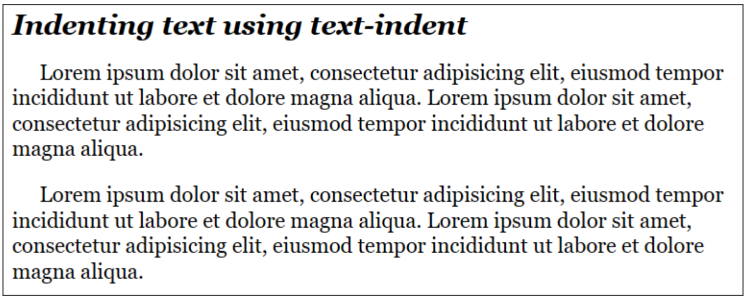
Figure 14.22 If You Align an Image with "vertical-align: top;" to the Top Edge of the Text, This Has Different Effects Than in the Lower Example with "float: left;"

14.1.12 Indenting Text Using "text-indent"

The CSS feature `text-indent` allows you to indent the first line of text with a positive value or drag it outward with a negative value. Such indentations are known mainly from books and the column typesetting of magazines, where the first line of a paragraph is indented to keep the reading flow smooth. On web pages, however, this kind of indentation occurs rarely. Here, the spacing from one paragraph to the next is more important, where the `p` element defaults to sufficient space from one paragraph to the next, and you can further adjust this using `margin`. In Figure 14.23, you can see such indentation of two paragraphs implemented with the following CSS statement:

```
...
.p-indent {  text-indent: 1.2em;  }
...
```

Listing 14.13 /examples/chapter014/14_1_12/css/style.css



Indenting text using text-indent

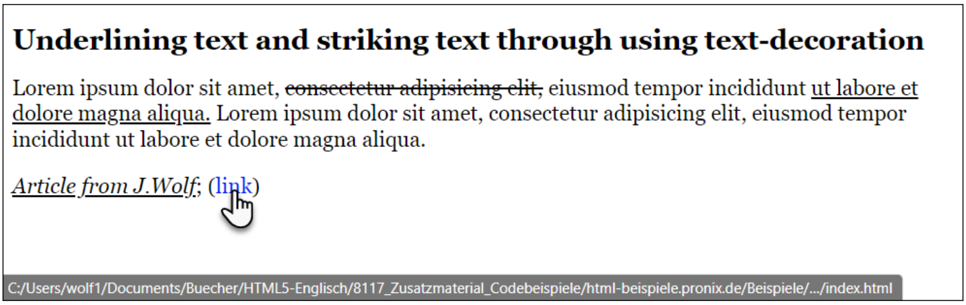
Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Figure 14.23 You Can Implement Text Indentation via the CSS Feature “text-indent”

14.1.13 Underlining Text and Striking Text Through Using “text-decoration”

To add an underscore to a text using CSS, you can use the CSS feature `text-decoration`. This allows you to draw a line under, above, or through the text. The values are `underline`, `overline`, and `line-through`. By using `none` (default setting), you can remove this decoration. Removing an underscore with `text-decoration: none;` can also be used to remove the underscore of a link (`<a>` tag).



Underlining text and striking text through using text-decoration

Lorem ipsum dolor sit amet, ~~consectetur adipisicing elit,~~ eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Article from J.Wolf; (link)

C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/html-beispiele.pronix.de/Beispiele/.../index.html

Figure 14.24 Underlining (or Undoing the Underlining) or Striking Text Through Using the CSS Feature “text-decoration”

In Figure 14.24, you can see the use of `text-decoration`, where body text was underlined using the `span` element and the author information by means of `text-decoration: underline;`. In addition, next to the author's name, the underline for the link was removed via `text-decoration: none;`, and text was struck through using the `span` element in the body text via `text-decoration: line-through`.

```
...
.underline { text-decoration: underline; }
.a-no-underline { text-decoration: none; }
.line-through { text-decoration: line-through; }
...
```

Listing 14.14 /examples/chapter014/14_1_13/css/style.css

Underlining Using "border-bottom"

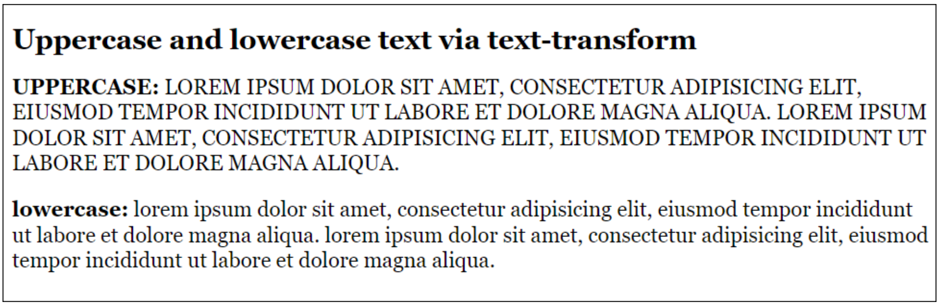
Because an underscore with `text-decoration` often crosses the letters g and y, you can use `border-bottom` for an underscore of texts that aren't links. This means that the letters y and g won't be crossed.

14.1.14 Uppercase and Lowercase Text via "text-transform"

You can use the CSS feature `text-transform` to control the case of the text. For this purpose, you can use the values `uppercase` (for uppercase letters), `lowercase` (for lowercase letters), and `capitalize` (first letter as uppercase). Again, the default setting is `none`.

The `capitalize` value, which is used to represent each first letter of a word as a capital letter, is usually used only in English for titles. In languages such as German, for example, this value is less interesting.

In Figure 14.25, you can see how the `h1` heading is displayed entirely in `uppercase` letters by means of `text-transform: uppercase;`. The paragraph, on the other hand, was set completely in lowercase via `text-transform: lowercase;`.



Uppercase and lowercase text via text-transform

UPPERCASE: LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISICING ELIT, EIUSMOD TEMPOR INCIDIDUNT UT LABORE ET DOLORE MAGNA ALIQUA. LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISICING ELIT, EIUSMOD TEMPOR INCIDIDUNT UT LABORE ET DOLORE MAGNA ALIQUA.

lowercase: lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Figure 14.25 Uppercase and Lowercase Text via "text-transform"

```
...
.uppercase { text-transform: uppercase; }
.lowercase { text-transform: lowercase; }
...
```

Listing 14.15 /examples/chapter014/14_1_14/css/style.css

Small Caps

If you're not looking for continuous uppercase letters, but small caps, you should take a look at the CSS feature `font-variant` from Section 14.1.6.

14.1.15 Adding Shadow to Text via "text-shadow"

A popular effect is to add a shadow to the text using the CSS feature `text-shadow`. The use of `text-shadow` is also quite convenient:

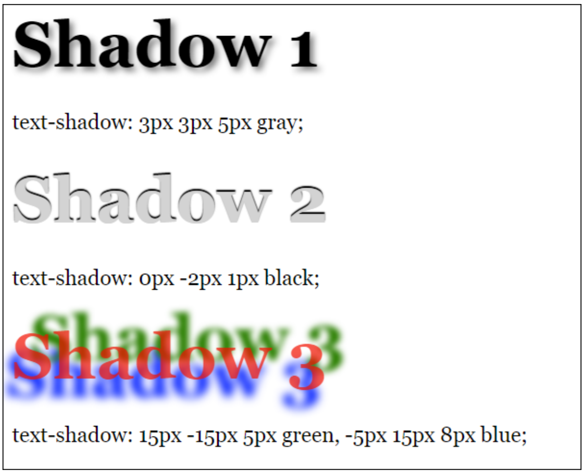
```
    text-shadow: 5px     /* Horizontal offset   */
                 5px     /* Vertical offset     */
                 4px     /* Gradient radius shadow */
                 gray;   /* Shadow color          */
```

In Figure 14.26, you can see some variants of shadows used via the following CSS statements for headings:

```
...
  .shadow-one { text-shadow: 3px 3px 5px gray; }
  .shadow-two {
    color: lightgray;
    text-shadow: 0px -2px 1px black;
  }
  .shadow-three {
    color: rgba(255, 0, 0, 0.7);
    text-shadow:
      15px -15px 5px green,
      -5px 15px 8px blue;
  }
...
```

Listing 14.16 /examples/chapter014/14_1_15/css/style.css

In the example, you can see that it's possible to use several shadows at once for one text (up to six). To do that, you simply need to list the shadows separated by commas.



Shadow 1

text-shadow: 3px 3px 5px gray;

Shadow 2

text-shadow: 0px -2px 1px black;

Shadow 3

text-shadow: 15px -15px 5px green, -5px 15px 8px blue;

Figure 14.26 Different Variants of Shadows

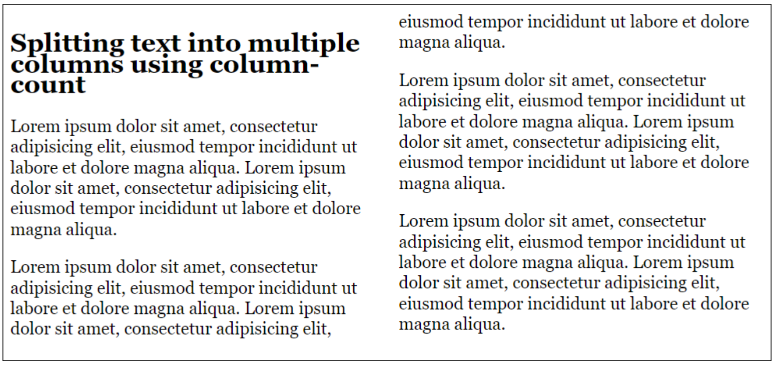
14.1.16 Splitting Text into Multiple Columns Using “column-count”

One very useful feature for typography is the ability to automatically split a text into a multicolumn set using the CSS feature `column-count` and without any manual work with JavaScript. This function is especially useful for wide screens, so that lines which are too long can be split up into columns, increasing the readability of the text.

Here’s how you can set up a two-column layout for the element used with `.column`:

```
.column {
  column-count: 2;
  column-gap: 1.5rem;
}
```

You can use `column-gap` to control the gap between the columns. In Figure 14.27, you can see the effect of these lines on an `article` element as a container with multiple `p` elements.



Splitting text into multiple columns using column-count

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Figure 14.27 The Multicolumn Set Has Been Applied to an <article> Element as a Container

Instead of `column-count`, you can also use `column-width` and specify a width for a column. Depending on the value you specify for the width, this will automatically create as many columns as there is space in the viewport of the web browser. When the web browser can no longer split the columns in width, it will make one single column out of it, for example:

```
.column {
  column-width: 250px;
  column-gap: 1.5rem;
}
```

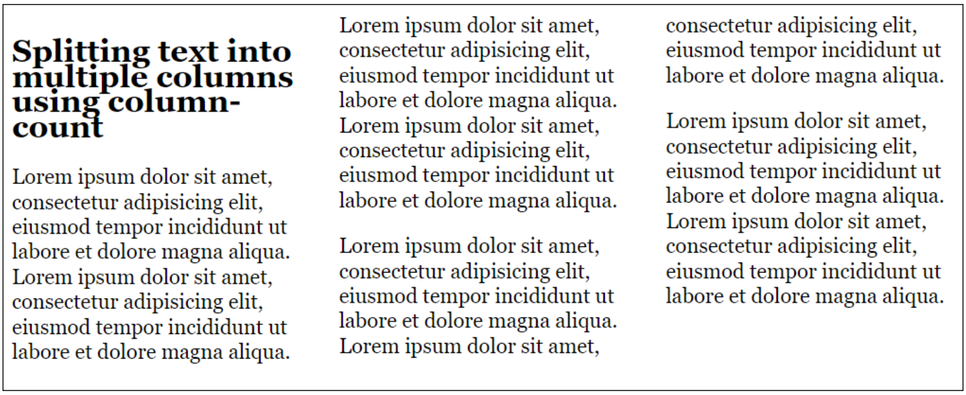
In Figure 14.28, the web browser has split the text into three columns of 250 pixels. In Figure 14.29, on the other hand, the browser window was reduced in size, and it was no longer possible to split the text into at least two columns of 250 pixels, so the text is now displayed in one column.

There's also a short notation available for the two properties `column-width` and `column-count` of the CSS feature `columns`:

```
.column {
  columns: 20em 2;
  column-gap: 1.5rem;
}
```

This way, you specify that two columns with a width of at least 20 em (320 pixels) should be used. If two columns of 20 em will no longer fit in the browser window, only one column will be used. This would be the case in the example if the viewport is less than 40 em or less than 640 pixels.

Other properties related to the multicolumn set are `column-rule`, which enables you to draw a line in the gap between columns, and `column-span`, which lets individual elements span multiple columns.



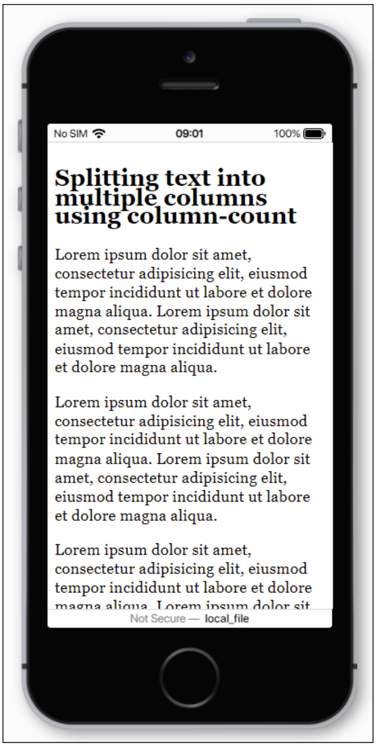
Splitting text into multiple columns using column-count

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Figure 14.28 Three Columns with 250 pixels



Splitting text into multiple columns using column-count

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit

Figure 14.29 If Two Columns No Longer Fit into the Width Specified with the CSS Property “column-width”, Only One Column Will Be Displayed

14.2 Designing Lists with CSS

You already know how to create lists in HTML from Chapter 4, Section 4.2. Now you’ll learn how to manipulate those lists using CSS. Strictly speaking, you can apply the CSS features `list-style-type`, `list-style-image`, and `list-style-position` to `<ul>` or `<ol>` for this purpose. A short notation for all three CSS features with `list-style` also exists. In the following sections, I’ll go into a little more detail about these CSS features for styling lists.

14.2.1 Customizing Bullet Points Using “list-style-type”

The CSS feature `list-style-type` allows you to specify the bullet selection of unordered lists with `<ul>` and the type of numbering ordered lists with `<ol>`.

For unordered lists with `<ul>`, the following values are available:

- **none**
  No bullet.

- **`disc`**
  Filled circle, also called *bullet character*, default setting.
- **`circle`**
  Empty circle as bullet character.
- **`square`**
  Square bullet sign.

For ordered lists with `<ol>` the following values are available, among others:

- **`decimal`**
  Numbering in the form 1., 2., 3., 4., 5., 6., and so on.
- **`decimal-leading-zero`**
  Numbering in the form 01., 02., 03., 04., and so on.
- **`lower-alpha` and `lower-latin`**
  Numbering in the form a., b., c., and so on.
- **`upper-alpha` and `upper-latin`**
  Numbering in the form A., B., C., and so on.
- **`lower-roman`**
  Numbering in the form i., ii., iii., iv., v., and so on.
- **`upper-roman`**
  Numbering like I., II., III., IV., V., and so on.
- **`none`**
  No numbering.

Numbering in Other Languages

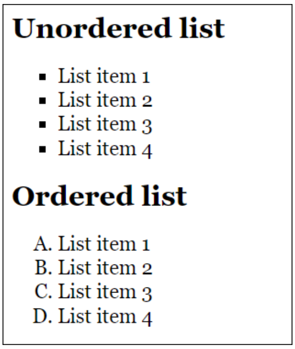
There are other numberings in other languages such as Armenian (`armenian`), Hebrew (`hebrew`), Georgian (`georgian`), or Japanese (`hiragana`).

In Figure 14.30, you can see how a square bullet (`square`) was used instead of the filled circle (with `disc`) for an unordered list, and alphabetical numbering in capital letters was used instead of decimal numbering (`decimal`) for the ordered list. Only the following two lines were used as CSS statements:

```
...
  ul { list-style-type: square; }
  ol { list-style-type: upper-alpha; }
...
```

Listing 14.17 /examples/chapter014/14_2_1/css/style.css

Feel free to experiment with the values for `<ul>` and `<ol>` listed previously.



Unordered list

- List item 1
- List item 2
- List item 3
- List item 4

Ordered list

A. List item 1
B. List item 2
C. List item 3
D. List item 4

Figure 14.30 Designing Bullets with “list-style-type”

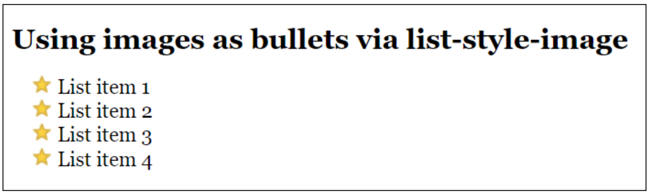
14.2.2 Using Images as Bullets via “list-style-image”

You can use the CSS feature `list-style-image` to add a custom graphic as an enumeration icon. The value you need to specify is `url(path)` with the path to a graphic.

In Figure 14.31, you can see such an example, where a simple graphic was used for the `ul` elements. The graphic was added with the following CSS line:

```
...
  ul { list-style-image: url("../graphic/stern.png"); }
...
```

Listing 14.18 /examples/chapter014/14_2/css/style.css



Using images as bullets via list-style-image

★ List item 1
★ List item 2
★ List item 3
★ List item 4

Figure 14.31 You Can Use a Graphic as a Bullet Point with the CSS Feature “list-style-image”

Using a Special Character Instead of a Graphic

As an alternative, you can also use a special character instead of a graphic. To do that, you simply need to set `list-style-type` to `none` and define a special character as a list icon via `li:before` using `content`:

```
...
.myul { list-style-type: none; }
.myul li:before { content: '\2713'; color: green; }
...
```

If you want to know what is represented by the character `'\2713'`, you should test this code (or look in */examples/chapter014/14_2/index2.html*). The advantage of using special characters instead of graphics is that you can adjust the size and color more easily.

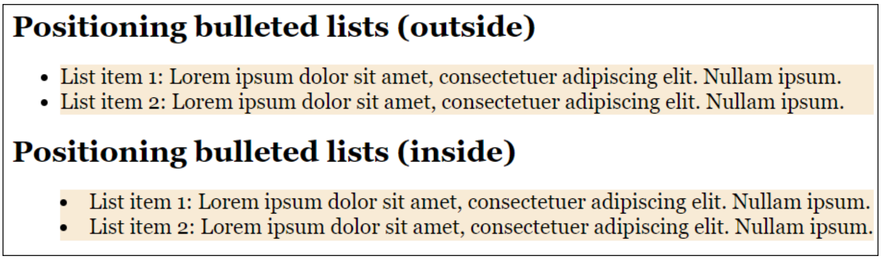
14.2.3 Positioning Bulleted Lists via “list-style-position”

You can use the CSS feature `list-style-position` to set whether the bullet should be inside or outside the box that contains the entries. The default behavior can be set using the `outside` value, which places the bullet point to the left of the text block. The counterpart to this value is `inside`, which places the bullet point inside the text block.

It’s best to look at Figure 14.32, where you can see the difference between `outside` and `inside`. In the example, the box with the `<li>` tag was assigned a gray background color. The CSS statements for that were written as follows:

```
...
.outside { list-style-position: outside; }
.inside { list-style-position: inside; }
...
```

Listing 14.19 /examples/chapter014/14_2_3/css/style.css



Positioning bulleted lists (outside)

- List item 1: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nullam ipsum.
- List item 2: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nullam ipsum.

Positioning bulleted lists (inside)

- List item 1: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nullam ipsum.
- List item 2: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nullam ipsum.

Figure 14.32 You Can Use “list-style-position” to Define Whether the Bullet Points Should Be outside (Default Setting) or inside the Box with the Entries

14.2.4 Short Notation “list-style” for Designing Lists

As is the case with several other CSS features, `list-style` is a short notation for the `list-style-type`, `list-style-image`, and `list-style-position` features, so you may be able to specify the shape, the graphic, and/or the bullet position in one go.

You can use any order and also enter only one or two entries. If you specify a graphic (`list-style-image`) with `url()`, the shape (`list-style-type`) will always be overwritten. `list-style-type` will only be used as an alternative if the graphic couldn’t be loaded. Even if the order is arbitrary, it’s recommended to maintain the order, as follows:

```
list-type: list-style-type list-style-position list-style-image;
```

This can simplify any troubleshooting work. Here's another example to clarify this:

```
ul { list-style: disc url(mybullet.png) inside; }
```

This short notation corresponds to the following entries:

```
ul {
  list-style-type: disc;
  list-style-image: url(mybullet.png);
  list-style-position: inside;
}
```

14.2.5 Creating Navigation and Menus via Lists

To create a navigation with CSS, lists are commonly used. In this section, I'll describe a simple way to implement a navigation using lists and CSS. The following HTML code with a list is used for this purpose:

```
...
<nav>
   <ul class="menu">
       <li class="logo"><img src="graphics/logo.png" height="30"></li>
       <li class="menu-item"><a href="#">Home page</a></li>
       <li class="menu-item"><a href="#">News</a></li>
       <li class="menu-item"><a href="#">About me</a></li>
       <li class="menu-item"><a href="#">Contact</a></li>
       <li class="menu-item"><a href="#">Privacy</a></li>
       <li class="menu-item button"><a href="#">Sign in</a></li>
   </ul>
</nav>
...
```

Listing 14.20 /examples/chapter014/14_2_5/index.html

The result of the list completely without styling is shown in Figure 14.33.

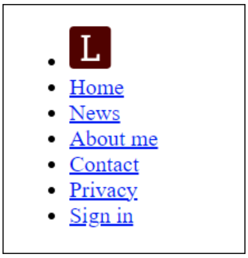


Figure 14.33 The Pure HTML Representation of the Navigation as a List

In Figure 14.34, on the other hand, I've already added basic styling to the individual elements in */examples/chapter014/14_2_5/css/style.css*. With regard to the lists, I've set `list-style-type` to `none` so that no bullet gets displayed.

```
...
ul {
    list-style-type: none;
}
...
```

Listing 14.21 /examples/chapter014/14_2_5/css/style.css

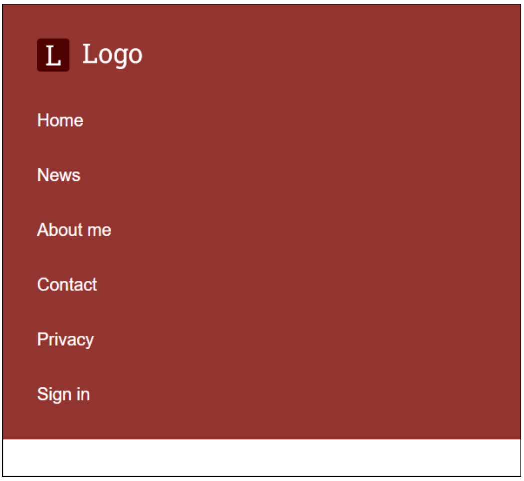


Figure 14.34 The List after a First Basic Styling

Creating the Mobile Navigation for the Smartphone

First of all, it's recommended to create the mobile navigation for smartphones. I chose a flexbox for this purpose because it allows me to lay out the menu and menu items without much effort. The important CSS lines for the mobile layout are as follows:

```
...
.menu {
    display: flex;
    flex-wrap: wrap;
    justify-content: space-between;
    align-items: center;
}

.menu-item.button {
    order: 1;
}
```

```
.menu-item {
    width: 100%;
    text-align: center;
    order: 2;
}
...
```

Listing 14.22 /examples/chapter014/14_2_5/css/style.css

By using `display: flex`, you make the `ul` element a flex container, while `align-items: center` allows you to center the `li` elements (here, `.menu-item`) vertically on the cross axis. `order` enables you to sort the order of the flexbox elements. In the mobile version, I can thus position the button (`order:1`) in front of the navigation elements of the menu (`order: 2`). You can extend the individual `li` menu items across the entire width using `width: 100%`. You can see the mobile version of the vertical navigation menu with lists in Figure 14.35.

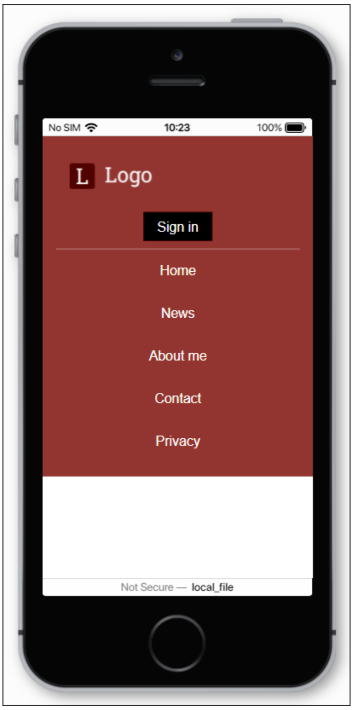


Figure 14.35 The Mobile Smartphone Version of the Vertical Navigation Menu with Lists

Creating the Vertical Navigation Menu for Tablets

I didn't change much for the vertical navigation menu for tablets. The logo can be extended using `flex: 1`. The button next to it, on the other hand, only gets as much space as it needs (`width: auto`). The other menu items remain at `width: 100%`, as in the smartphone version and thus still extend across the entire width, placing each element in its own line.

```
...
@media all and (min-width: 37.5em) {
    .logo {
        flex: 1;
    }
    .menu-item.button {
        width: auto;
        border-bottom: 0;
    }
}
...
```

Listing 14.23 /examples/chapter014/14_2_5/css/style.css

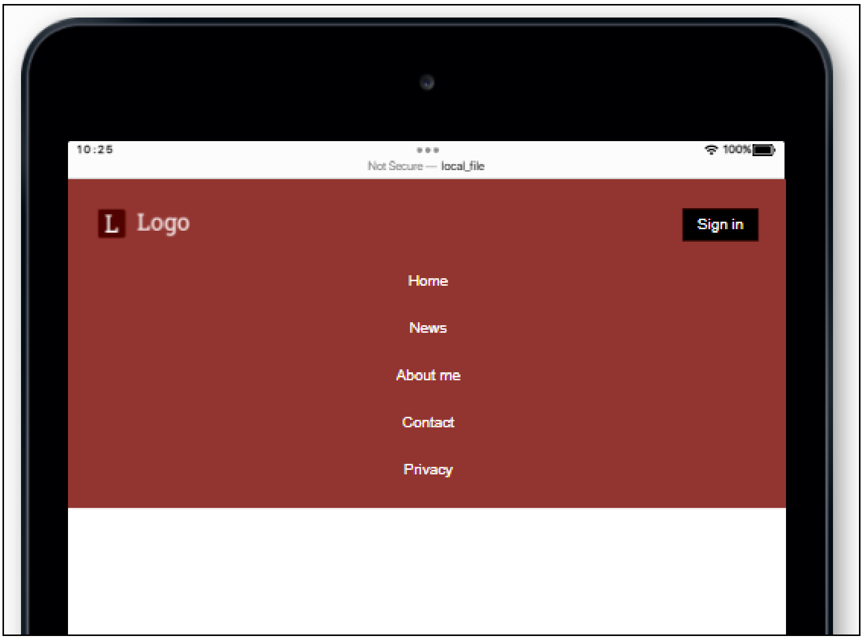

10:25
Not Secure — local_file
100%
L Logo
Sign in
Home
News
About me
Contact
Privacy

Figure 14.36 The Tablet Version of the Vertical Navigation with List Items

Expandable Menu with JavaScript for Mobile Versions

I didn't include an expandable menu for the mobile versions here because I would have had to anticipate JavaScript at this point. Instead, I created the example using a very simple expandable menu with jQuery, which you can find in */examples/chapter014/14_2_5/index2.html*.

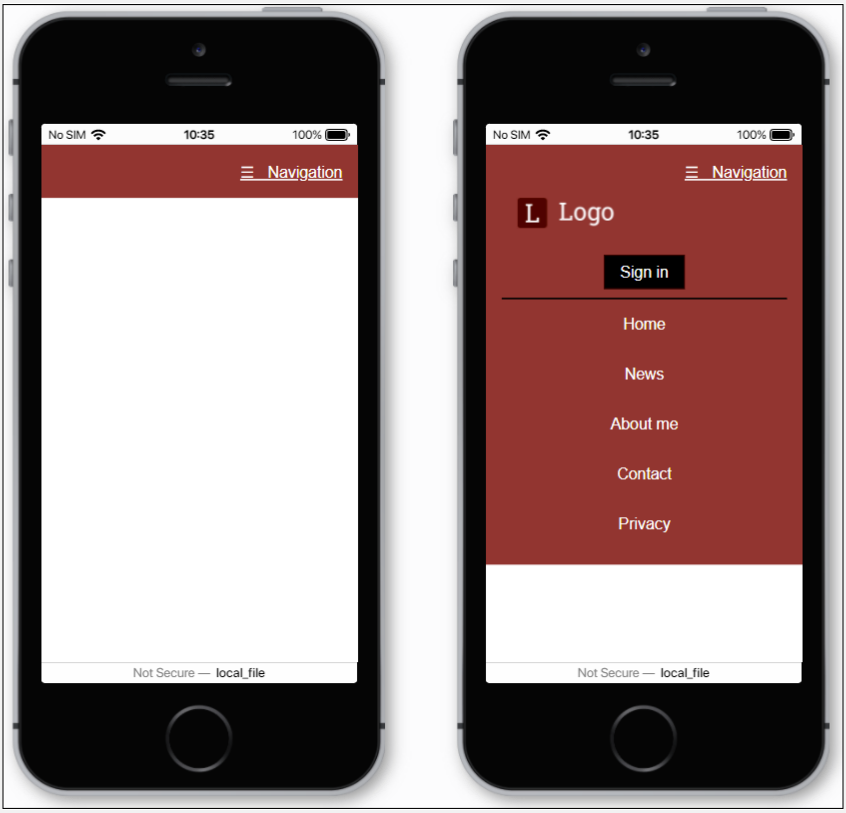

No SIM
10:35
100%
☰ Navigation
Not Secure — local_file
No SIM
10:35
100%
☰ Navigation
L Logo
Sign in
Home
News
About me
Contact
Privacy
Not Secure — local_file

Figure 14.37 A Simple Expandable Menu with jQuery

Creating the Vertical Navigation Menu for Desktops

For the desktop version, I keep it very simple and set all menu items to `width: auto`, making them share the space behind the logo, which still uses `flex: 1`. If you want to provide all elements with equal space in the horizontal navigation, you could do this by setting the CSS feature `flex` to 1 in the `menu-item` class as well.

```
...
@media all and (min-width: 60em) {
    .menu-item {
        width: auto;
    }
    .logo {
        order: 0;
    }
    .menu-item {
        order: 1;
    }
    .button {
        order: 2;
    }
    .menu li.button {
        padding-right: 0;
    }
}
```

Listing 14.24 /examples/chapter014/14_2_5/css/style.css

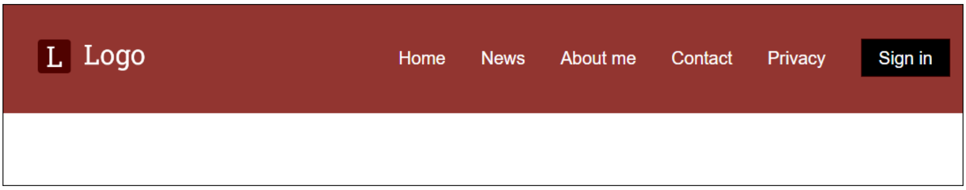
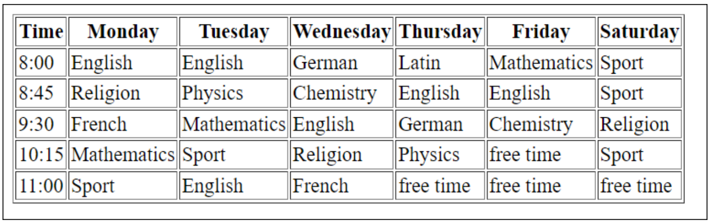


Figure 14.38 The Desktop Version of the Vertical Navigation with List Elements

14.3 Designing Appealing Tables with CSS

Tables are commonly used to present data in a clear way. Thanks to CSS, you can make a boring HTML table more attractive and appealing. You already know many CSS features for designing a table from other sections (but not yet in connection with tables). At this point, I only want to mention the CSS features that are particularly useful in the context of tables. To demonstrate this, we'll style the timetable shown in Figure 14.39 from pure HTML with CSS in the following sections.



Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	Sport
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French	Mathematics	English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	Sport
11:00	Sport	English	French	free time	free time	free time

Figure 14.39 A Boring Table in Pure HTML

14.3.1 Creating Fixed-Width Tables

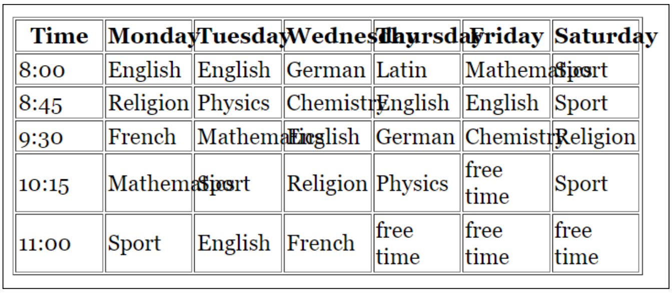
From Chapter 5, Section 5.1, you know that if `width` is too small, the contents of table cells without special specifications will take up the space they just need to get displayed. It doesn't matter if you want to reduce the table width explicitly via `width` to `500px`, `300px`, or `50px`, the result will always look as shown in Figure 14.39 because the table always tries to fit to the content.

It's still possible to use the CSS feature `table-layout` to assign a specified and fixed width to the table elements. The default value of `table-layout` is `auto` and provides the view you're used to. If, on the other hand, you use the value `fixed` for the CSS feature `table-layout`, then exactly the width you specified via `width` will be used. If the content doesn't fit into the table cell anymore, it will be wrapped, truncated (depending on the value of the `overflow` feature), or just written beyond the cell boundary. For example, with reference to Figure 14.39, I've limited the width of the table to 500 pixels and fixed the table layout as follows:

```
...
.table-fixed {
  table-layout: fixed;
  width: 500px;
}
...
```

Listing 14.25 /examples/chapter014/14_3_1/css/style.css

You can see the (not really appealing) result of this fixation in Figure 14.40.



Time	Monday	Tuesday	Wednes	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathema	Sport
8:45	Religion	Physics	Chemistr	English	English	Sport
9:30	French	Mathema	English	German	Chemistr	Religion
10:15	Mathema	Sport	Religion	Physics	free time	Sport
11:00	Sport	English	French	free time	free time	free time

Figure 14.40 When You Use "table-layout: fixed;", Then No More Consideration Is Given to the Content

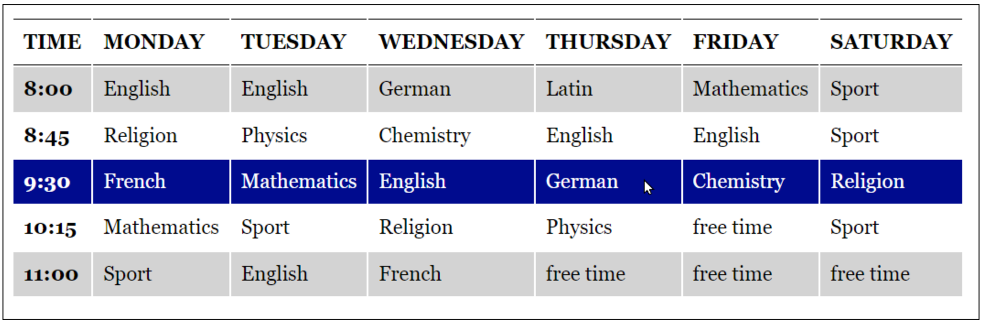
14.3.2 General Recommendation: Designing Appealing Tables with CSS

Because you may also look in this book for a general recipe for formatting a table properly with CSS, I'll discuss this briefly here and give you a few tips on how I would go about it. I summarize a table in HTML as usual. Then the table is assigned a width via `width`. I format the table header differently from the rest of the table cells. For a better overview, I color the individual table rows alternately with two different colors. Almost all table cells get padding by means of `padding`. When it makes sense, I use the CSS pseudo-class `:hover`, which highlights a table row when the user hovers over it with the mouse. In summary, a basic layout of a table with CSS usually looks like this in my case:

```
...
table {
  width: 700px;
}
th {
  padding: 0.5em;
  text-transform: uppercase;
  border-top: 1px solid black;
  border-bottom: 1px solid black;
  text-align: left;
}
tr:nth-child(even) { background: lightgray; }
td:nth-child(1) {
  font-weight: bold;
  width: 100px;
}
td { padding: 0.5em; }
tr:hover {
  background: darkblue;
  color: white;
}
...
```

Listing 14.26 /examples/chapter014/14_3_2/css/style.css

You can see the result of these few lines of CSS used to design a complete table in Figure 14.41.



TIME	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY
8:00	English	English	German	Latin	Mathematics	Sport
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French	Mathematics	English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	Sport
11:00	Sport	English	French	free time	free time	free time

Figure 14.41 The Basic Formatting of an HTML Table with CSS Is Done with a Few Lines

14.3.3 Collapsing Borders for Table Cells Using “border-collapse”

The CSS feature `border-collapse` allows you to specify whether the borders of individual cells are displayed separately (`border-collapse: separate;`, default setting) or collapsed (`border-collapse: collapse;`). In Figure 14.42, you can see the use of the `separate` value, and in Figure 14.43 the use of `collapse`.

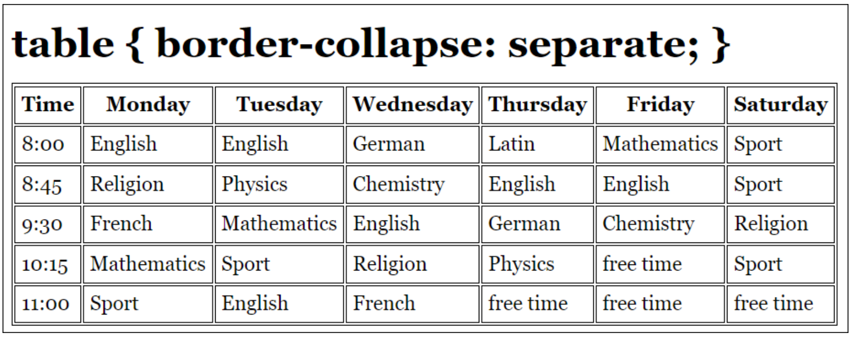


table { border-collapse: separate; }

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	Sport
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French	Mathematics	English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	Sport
11:00	Sport	English	French	free time	free time	free time

Figure 14.42 Frames of Adjacent Elements Are Displayed Separately with “border-collapse: separate;” (= default setting)

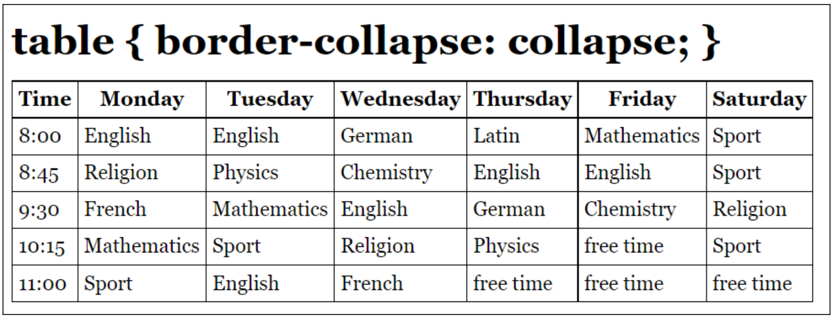


table { border-collapse: collapse; }

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	Sport
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French	Mathematics	English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	Sport
11:00	Sport	English	French	free time	free time	free time

Figure 14.43 Due to “border-collapse: collapse;”, the Borders of the Adjacent Elements Collapse (Example in /examples/chapter014/14_3_3/index.html)

14.3.4 Setting the Spacing between Cells via “border-spacing”

If the CSS feature of `border-collapse` isn’t equal to `collapse`, you can set the spacing of adjacent cells by using `border-spacing`. The specification for this is mostly in pixels, and you can also specify separately the values for a horizontal and vertical spacing. Compared to Figure 14.42, for the example in Figure 14.44, we used `border-spacing: 5px 10px;`, where the spacing in horizontal direction is 5 pixels and in the vertical direction 10 pixels to the adjacent table element.

“border-spacing: 0px;”

If you set `border-spacing` to `0`, the borders won’t collapse as they do with `border-collapse: collapse;`, but the cells will be positioned exactly next to each other.

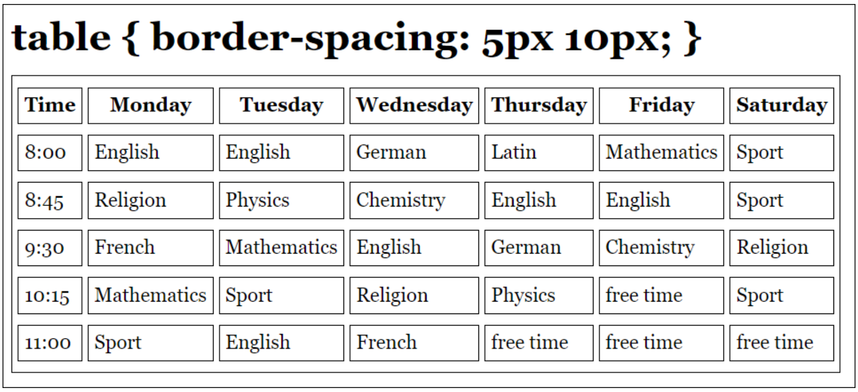


table { border-spacing: 5px 10px; }

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	Sport
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French	Mathematics	English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	Sport
11:00	Sport	English	French	free time	free time	free time

Figure 14.44 You Can Adjust the Spacing between the Table Cells via "border-spacing" (Example in /examples/chapter014/14_3_4/index.html)

14.3.5 Displaying Empty Table Cells Using "empty-cells"

If you have an empty table cell, you can use the CSS feature `empty-cells` to specify whether or not you want to draw a border around it. By default, the web browser usually displays a border around an empty table cell, as shown in Figure 14.45, which corresponds to the value `show` (`empty-cells: show;`).

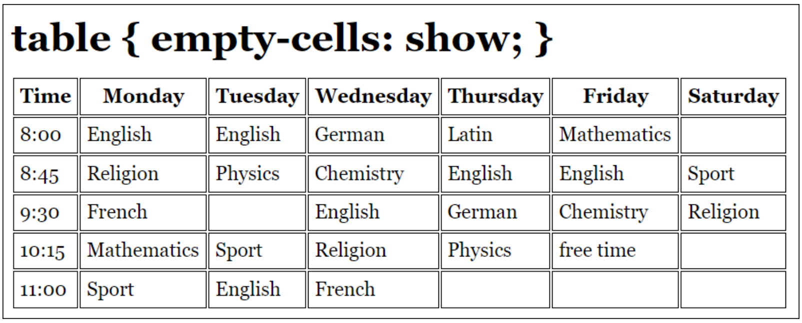


table { empty-cells: show; }

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French		English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	
11:00	Sport	English	French			

Figure 14.45 Showing Borders for Empty Cells Is the Default Setting, Which Can Also Be Written as "empty-cells: show;"

If you don't want a border to be drawn around a table cell with no content, you just need to assign the `hide` value to the `empty-cells` feature, as was done in the example in Figure 14.46. If you use the `collapse` value and not `separate` for the CSS feature `border-collapse`, the `empty-cells: hide;` specification will be ignored.

Empty Cells

A line feed, space, or tab feed is considered invisible content. On the other hand, if you write an enforced blank space with `&nbsp;` in the table cell, this will be considered visible content, and a border will be drawn around it if you've used `empty-cells: hide;`.

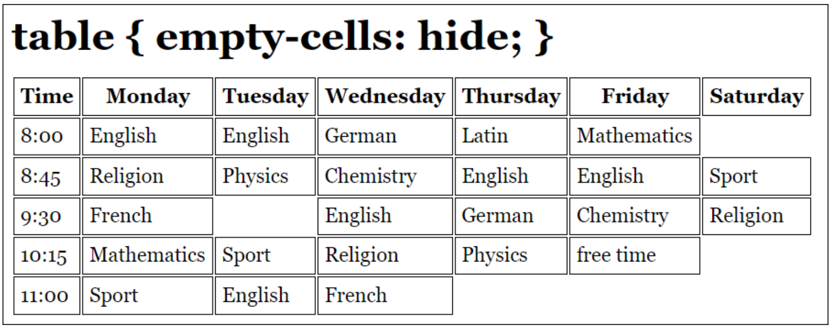


table { empty-cells: hide; }

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French		English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	
11:00	Sport	English	French			

Figure 14.46 If You Want to Hide the Border for Empty Cells, You Can Do This by Using “empty-cells: hide;” (Example in /examples/chapter014/14_3_5/index.html)

14.3.6 Positioning Table Captions via “caption-side”

As you may already recognize from the title of the CSS feature `caption-side`, this feature sets the position of the table caption you used with the `caption` element. The default is usually a display above the table, which corresponds to the `top` value for `caption-side`. By means of `bottom`, you can position the caption below the table, as I did in Figure 14.47.

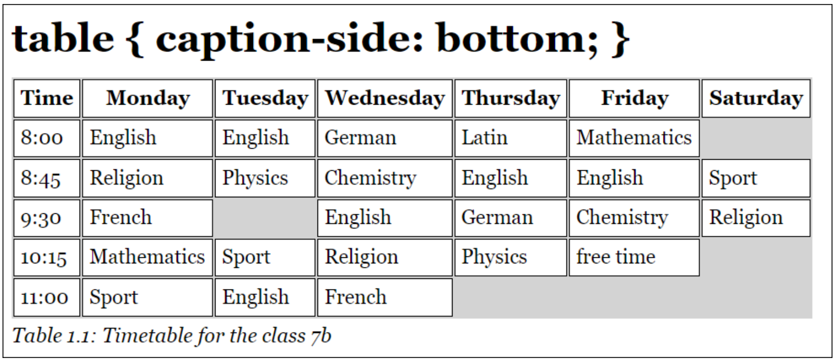


table { caption-side: bottom; }

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
8:00	English	English	German	Latin	Mathematics	
8:45	Religion	Physics	Chemistry	English	English	Sport
9:30	French		English	German	Chemistry	Religion
10:15	Mathematics	Sport	Religion	Physics	free time	
11:00	Sport	English	French			

Table 1.1: Timetable for the class 7b

Figure 14.47 The Table Caption with <caption> Has Been Moved to the Bottom with “caption-side: bottom;” (Example in /examples/chapter014/14_3_6/index.html)

14.4 Adjusting Images and Graphics Using “width” and “height”

Basically, you already know the CSS features you can use to customize images and graphics with CSS if you’ve read the book from the beginning. You can also set the size of the images using the CSS features, `width` and `height`. For example:

```
...
.large {
    width: 325px;
    height: 267px;
}

.medium {
    width: 225px;
    height: 184px;
}

.small {
    width: 125px;
    height: 103px;
} }
```

Listing 14.27 /examples/chapter014/14_4_1/css/style.css

For example, with these three classes, you can conveniently output the same image in three different sizes. Instead of `width` and `height`, you just need to specify the corresponding class name for the `img` elements. You can see the result of the following HTML lines in Figure 14.48:

```
...
    <img src="cover.png" alt="Computer History" class="small" />
    <img src="cover.png" alt="Computer History" class="medium" />
    <img src="cover.png" alt="Computer History" class="large" />

...
```

Listing 14.28 /examples/chapter014/14_4_1/index.html

You already know how to align images, namely by giving the CSS feature `float` the value `left` or `right`. If you also want to center an image, you merely need to make a block element out of the `img` element via `display: block;`, then you can use `margin: 0px auto;` or `text-align: center;` to center-align the image. Here are three more classes you can use to align or center images with CSS:

```
...
  img.align-left {
    float: left;
    margin: 0 0.6em 0.3em 0;
  }
  img.align-right {
    float: right;
    margin: 0 0 0.3em 0.6em;
  }
```

```
img.align-center {
  display: block;
  margin: 0px auto;
}
```

Listing 14.29 /examples/chapter014/14_4_1/css/style2.css

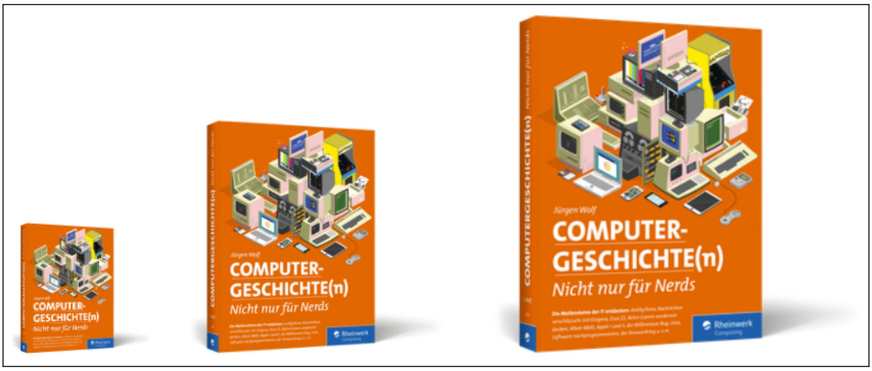
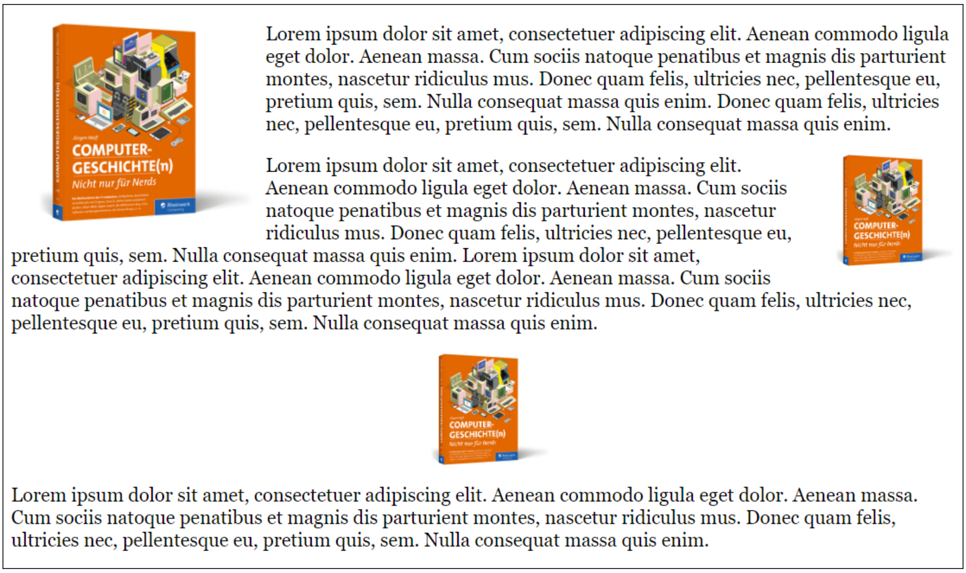


Figure 14.48 One and the Same Image Was Put into a Class with the CSS Features “width” and “height” and Used in Different Sizes

You can combine the classes created in this way with the classes for the appropriate size and have the images output in the appropriate size and orientation for your website in no time at all. You can view the result of the following HTML code in Figure 14.49:

```
...
  <p>
    <img src="cover.png" alt="Computer History"
         class="align-left medium">
    Lorem ipsum ...
  </p>
  <p>
    <img src="cover.png" alt="Computer History"
         class="align-right small">
    Lorem ipsum ...
  </p>
  <p>
    <img src="cover.png" alt="Computer History"
         class="small align-center">
    Lorem ipsum ...
  </p>
...
```

Listing 14.30 /examples/chapter014/14_4_1/index2.html



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Figure 14.49 Graphics Resized and Aligned with CSS

14.5 Transforming Elements with CSS

With CSS, it's also possible to change the position of HTML elements using the CSS feature `transform`. The possible actions are movements via `translate()`, an enlargement or reduction using `scale()`, a rotation by means of `rotate()`, the skewing of elements via `skew()`, and a distortion with `matrix()`. These transformations are supported by all currently available web browsers.

Although these transformations can be applied to other HTML elements as well, for demonstration purposes, we'll use images that have been placed side by side using a flexbox. You can see the starting point for the planned transformations in Figure 14.50.

```
...
<h1>Transforming images with CSS</h1>
<ul>
    <li>
        <img src="images/image01.jpg" alt="China photography tour 2019">
    </li>
    <li>
        <img src="images/image02.jpg" alt="China photography tour 2019">
    </li>
    <li>
        <img src="images/image03.jpg" alt="China photography tour 2019">
    </li>
    <li>
        <img src="images/image04.jpg" alt="China photography tour 2019">
    </li>
    <li>
        <img src="images/image05.jpg" alt="China photography tour 2019">
    </li>
...
</ul>
...
```

Listing 14.31 /examples/chapter014/14_5/index.html

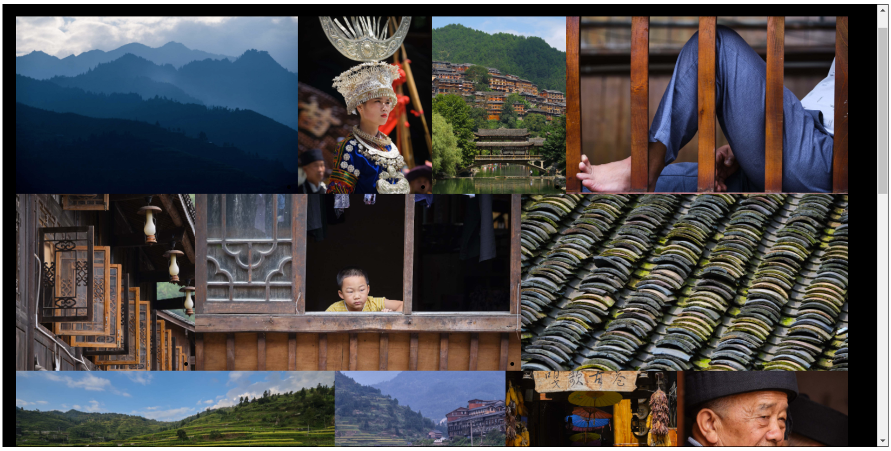


Figure 14.50 These Images Are Supposed to Be Transformed When Users Hover over Them (":hover")

14.5.1 Scaling HTML Elements via "transform: scale()"

You can enlarge (or reduce) elements using the CSS feature `transform` and the CSS function `scale()`. A value of `scale(1.0)` has no effect. If you use `scale(1.25)`, the element will be enlarged by a factor of 1.25. Similarly, if you specify `scale(0.75)`, an image will be `scaled` down by a factor of 0.75. The surrounding elements aren't affected by an enlargement or reduction and remain firmly in position.

Here's an example in which an image is to be enlarged by a factor of 1.25 when hovering. The results of the following CSS lines are shown in Figure 14.51:

```
...
img:hover {
  transform: scale(1.25);
}
```

Listing 14.32 /examples/chapter014/14_5_1/css/style.css

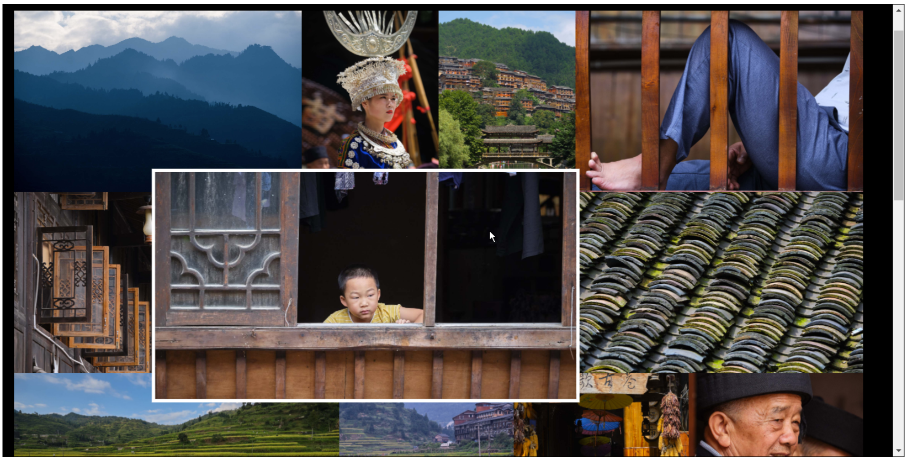


Figure 14.51 The Images Are Enlarged by a Factor of 1.25 When You Move the Cursor over Them (":hover")

14.5.2 Rotating HTML Elements Using "transform: rotate()"

When you use `rotate()`, the respective element gets rotated by a specified number of degrees. The specification is in the form of `rotate(15deg)`, which rotates the element clockwise by 15 *degrees*. A negative value rotates the element counterclockwise. You can see the result of the following CSS lines in Figure 14.52.

```
...
.trans a img:hover {
  transform: rotate(15deg);
}
```

Listing 14.33 /examples/chapter014/14_5_2/css/style.css

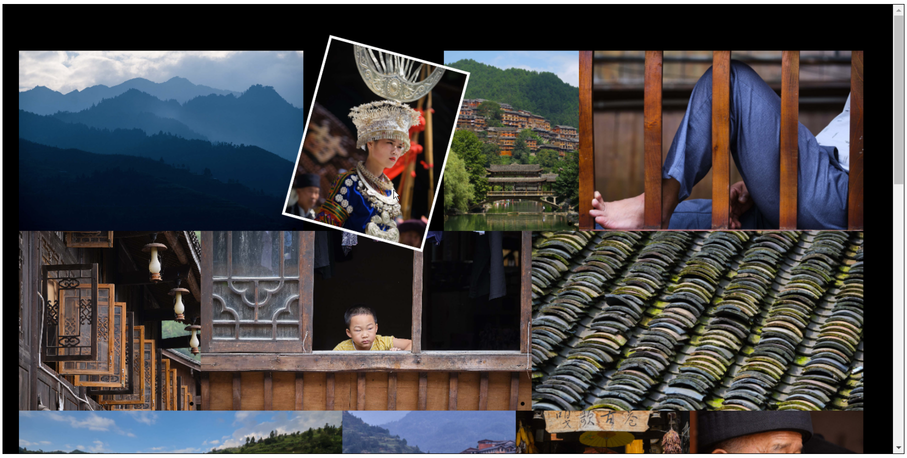


Figure 14.52 A Rotation on Mouseover Using "transform: rotate()"

14.5.3 Skewing HTML Elements Using “transform: skew()”

The `skew()` function can be used to skew an HTML element around the x-axis and y-axis. Here, too, two values are expected as degrees. The first value indicates the skew around the x-axis and the second one around the y-axis. For example, you can use `skew(5deg, 10deg)` to rotate the element 5 degrees around the x-axis and 10 degrees around the y-axis. You can view the results of the following CSS lines in Figure 14.53.

```
...
.trans a img:hover {
  transform: skew(5deg, 10deg);
}
```

Listing 14.34 /examples/chapter014/14_5_3/css/style.css

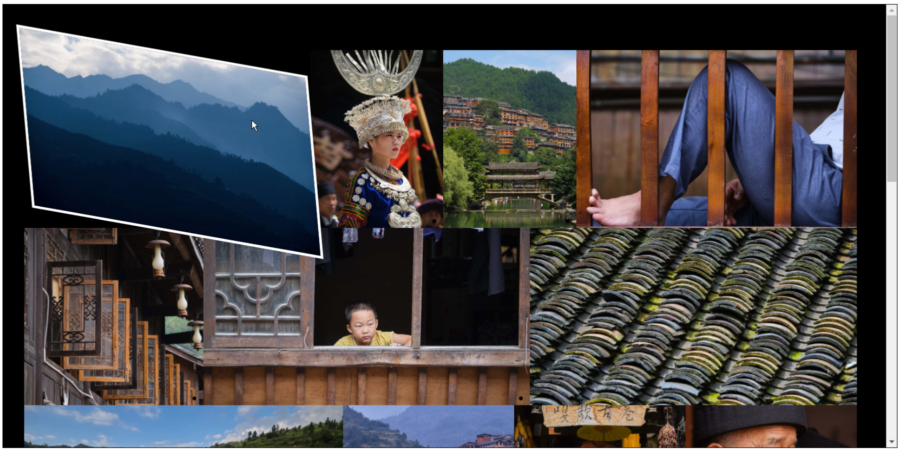


Figure 14.53 Skewing HTML Elements via “transform: skew()”

Additional Functions via “skewX()” and “skewY()”

If you want to skew an HTML element only around the x-axis or y-axis, the corresponding functions for these actions are `skewX()` and `skewY()`, respectively.

14.5.4 Moving HTML Elements Using “transform: translate()”

The `translate()` function enables you to move elements. For this purpose, you also need to specify two values to indicate by how much you want to move the element along the x-axis and y-axis. A specification such as `translate(10px, 20px)` moves the element 10 pixels to the right along the x-axis and 20 pixels down along the y-axis. Negative values move the element in the other direction. The following CSS snippet causes the movement shown in Figure 14.54 when you halt the mouse cursor on the image:

```
...
#trans a img:hover {
  transform: translate(30px, 20px);
}
```

Listing 14.35 /examples/chapter014/14_5_4/css/style.css

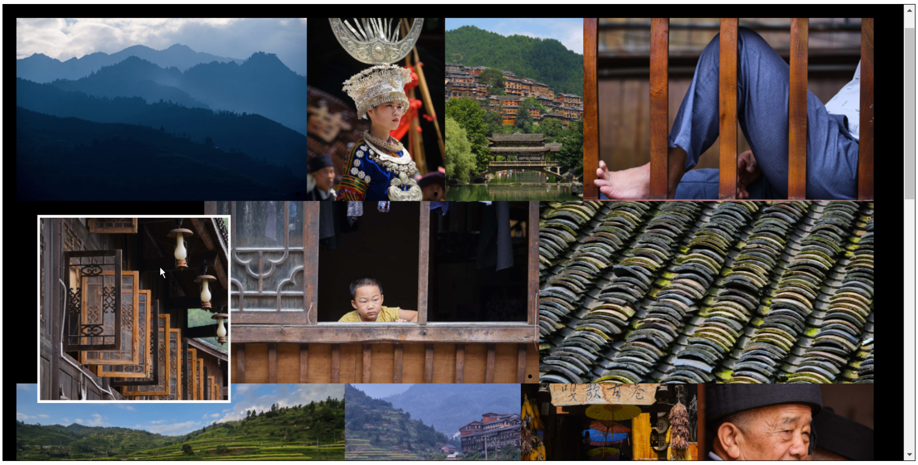


Figure 14.54 Moving HTML Elements via “transform: translate()”

14.5.5 Combining Different Transformations

It’s possible to combine several functions for the purpose of transforming. To do that, you only need to specify the respective functions separated by a space. Here’s a simple example of this, in which an element gets enlarged by a factor of 1.25 and rotated clockwise by 10 degrees (see Listing 14.36).

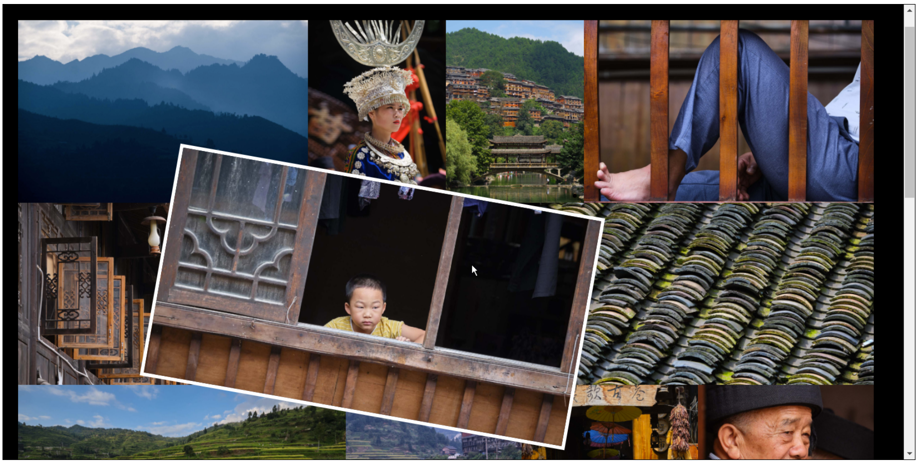


Figure 14.55 The Element Was Enlarged and Rotated

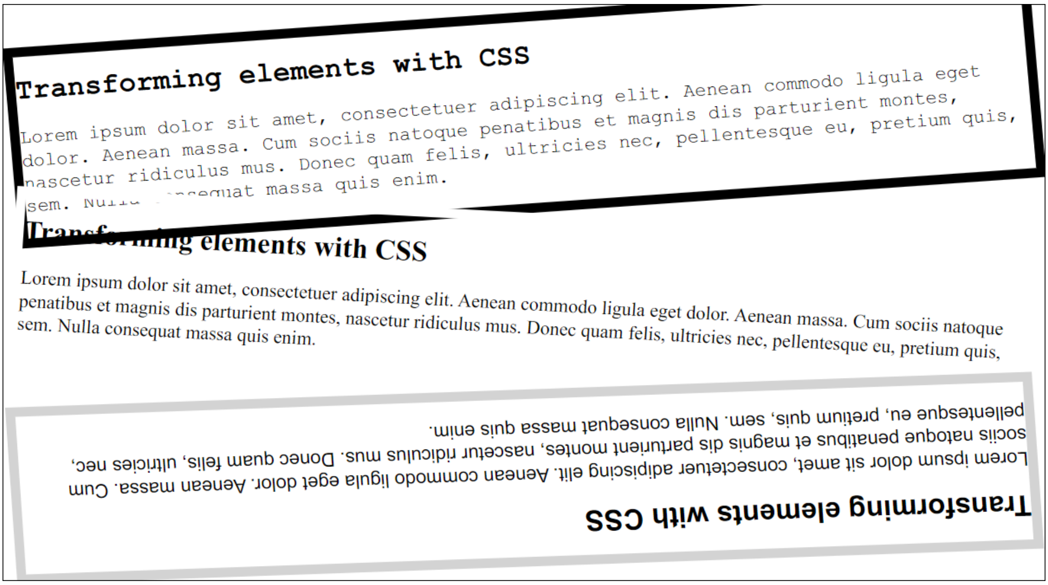
```
...
.trans a img:hover {
  transform: scale(1.25) rotate(10deg);
}
```

Listing 14.36 /examples/chapter014/14_5_5/css/style.css

14.5.6 Other HTML Elements

By the way, the transform functions presented here aren't limited to images or graphics and can be used for other HTML elements as well. Likewise, you can transform the HTML elements at any time and don't need to use hovering with the mouse to do so, although this is what people tend to prefer.

In Figure 14.56, for example, the articles were rotated or skewed via the `rotate()` and `skew()` functions. Of course, this isn't always useful, but my point is to show that these functions can be applied to other HTML elements too.



Transforming elements with CSS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget
dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes,
nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis,
sem. Nul sequat massa quis enim.

Transf ming elements with CSS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque
penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis,
sem. Nulla consequat massa quis enim.

Transforming elements with CSS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum
sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec,
pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim.

Figure 14.56 Other HTML Elements Can Also Be Transformed. Here, <article> Elements Were Rotated or Skewed (Example in /examples/chapter014/14_5_6/index.html)

Other Functions for Transforming

I haven't mentioned `transform-origin(x, y)` yet, which allows you to move the origin point from the element to be transformed.

14.6 Creating Transitions with CSS

In the examples with mouse hovering and using `transform`, the transition of the effects was a bit messy after all. For example, in */examples/chapter014/14_5_5/index.html*, which is shown in Figure 14.55, if you place the mouse cursor over the image, the graphic gets immediately scaled up and rotated, just like a light switch, and then immediately returned to its normal position when you move the mouse somewhere else. If you find this transition a bit too abrupt, you can soften it using the CSS feature `transition`. This doesn't require much effort at all, as shown in the following CSS snippet:

```
...
img {
    max-height: 100%;
    min-width: 100%;
    object-fit: cover;
    vertical-align: bottom;
    transition: all 1s ease;
}

img:hover {
    transform: scale(1.25) rotate(10deg);
    border: 4px white solid;
    transition: all 1s ease;
}
```

Listing 14.37 /examples/chapter014/14_6/css/style.css

This example corresponds to the one in Figure 14.55, except that the image gets rotated and enlarged slowly (here exactly within 1 second) while you hover the mouse cursor over it. To ensure that the image gets moved back to the normal position just as slowly and not abruptly, the normal position was also defined using the same `transition` statement. More actions weren't necessary so that now there's no longer a jerky effect in conjunction with `transform` when you hover over the image. The best thing to do is to try out this example for yourself.

When looking at the `transition` feature, you'll notice that three values have been used here: `all`, `1s`, and `ease`. Strictly speaking, `transition` is a short notation of the following features:

- **`transition-property`**
  Allows you to specify the property to be animated during the transition. With `all`, you can specify that all properties are animated. You can also specify only individual properties here, such as `background` for animating.

- **`transition-duration`**
  This specifies the duration of the transition in seconds. You can also determine the fraction of a second with, for example, `0.2s`, which is two tenths of a second.
- **`transition-timing-function`**
  This specification is a kind of temporal course of the transition. For example, the `ease` specification used here means that the transition starts slowly, speeds up a bit in the middle, and ends slowly again. There are several such temporal progressions such as `linear`, `ease-in`, `ease-out`, or `ease-in-out`, which you can try for yourself.

Optionally, you could use `transition-delay`, which adds a delay at the start of the transition. Consequently, you could alternatively write the short notation of `transition` used previously as follows:

```
...
img:hover {
  ...
  transition-property: all;
  transition-duration: 1s;
  transition-timing-function: ease;
}
...
```

Examples and Overview of "transition"

For some great demonstrations and examples of transitions using `transition`, you can visit *http://css3.bradshawenterprises.com/transitions/*.

14.7 Styling HTML Forms with CSS

While you've seen HTML forms in action, I haven't yet described how to design forms with CSS. The following isn't meant to be the ultimate way, but rather a creative suggestion on how you can go about it. There are certainly countless ways to design a form with CSS. Here, I'll show you one of them.

Less Is More!

CSS now provides an extremely wide range of options for designing forms, which are only briefly touched on here. After these sections on forms, you should take a look at the many examples of HTML forms on the internet.

Please keep in mind that, despite the wide range of design options, forms are real functional elements of a website, and when designing them, you want to make sure that these elements remain recognizable as what they are intended for. A form is usually used to submit entered data to the web server via a web browser.

14.7.1 Neatly Structuring an HTML Form

The first step should be to create a structure with all necessary form elements in HTML. I decided to use the example of a simple HTML form that submits feedback or a simple message. Here's the HTML framework for it:

```
...
<form id="myForm" method="post">
  <fieldset>

    <div>
      <label for="name">Name:</label>
      <input type="text" name="name" id="name"
             placeholder="Your name">
    </div>
    <div>
      <label for="fname">First name:</label>
      <input type="text" name="fname" id="fname"
             placeholder="Your first name">
    </div>
    <div>
      <label for="mail">Email:</label>
      <input type="email" name="mail" id="mail"
             placeholder="Email address" required>
      <label for="mail"></label>
    </div>
    <div>
      <label for="born">Year of birth:</label>
      <input type="number" name="born" id="born"
             min="1920" max="2015" value="1990">
    </div>
    <div class="form_radio">
      <label>Gender:</label>
      <input type="radio" name="gender" id="male"
             value="male" class="nobreak">
      <label for="male" class="nobreak">Male</label>
      <input type="radio" name="gender" id="female"
             value="female" class="nobreak">
      <label for="female" class="nobreak">Female</label>
    </div>
    <div>
      <label for="nachricht">Your message:</label>
      <textarea name="message" id="message"
                placeholder="Enter message here..." rows="8"
                required>
      </textarea>
      <label for="message"></label>
    </div>
    <div>
      <input type="checkbox" name="reply" id="reply"
             value="reply" class="nobreak">
      <label for="reply" class="nobreak">
         GDPR consent (<a href="#">Privacy policy</a>)
      </label>
    </div>
    <div>
      <input name="submit" type="submit" value="Submit" class="nobreak">
      <input name="Reset" type="reset" value="Reset" class="nobreak">
    </div>
    <p>(×) = Input required</p>
  </fieldset>
</form>
...
```

Listing 14.38 /examples/chapter014/14_7/index.html

There isn't much more that needs to be said about this HTML form; you learned about all the individual elements and attributes in detail in Chapter 7.

If you want to make CSS life easier with HTML forms, you can place all input fields with their associated labels between `<div>` and `</div>` right away, as shown in the example. This will display the related elements directly in one line; you can see the difference in Figure 14.57 (without `<div>`) and in Figure 14.58 (with `<div>`).

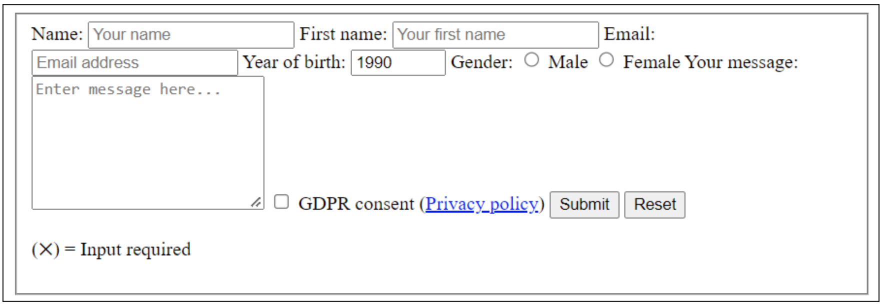
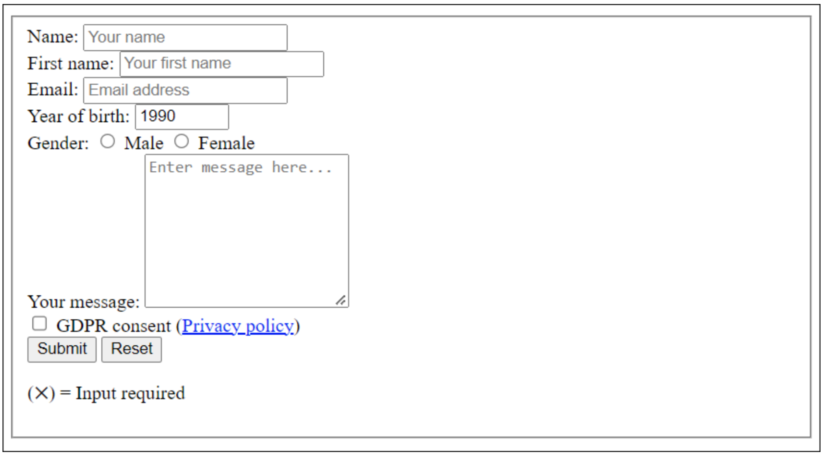

Name: Your name
First name: Your first name
Email:
Email address
Year of birth: 1990
Gender: ○ Male ○ Female Your message:
Enter message here...
☐ GDPR consent (Privacy policy)
Submit
Reset
(×) = Input required

Figure 14.57 The Form without the <div> Elements

Likewise, it can only be recommended to use `<fieldset>` and `<label>` because these elements can be very useful for design tasks.

Between `<fieldset>` and `</fieldset>`, you can group logically matching areas of form elements and at the same time have another approach to use CSS to design this area.



Name: Your name
First name: Your first name
Email: Email address
Year of birth: 1990
Gender: ○ Male ○ Female
Enter message here...
Your message:
☐ GDPR consent (Privacy policy)

(X) = Input required

Figure 14.58 Here's the Form with the <div> Elements

The same applies to `<label>` because not only do you use it to create a connection to the form element with the HTML attribute `for`, which means that when you click on the label, you immediately activate the form element, but you also get a kind of grid in which elements are displayed next to each other as in a table by specifying `<label>` and another form element such as `<input>`. Furthermore, you could design a `<label>` with an ID or class separately, which is more difficult with an empty text. Finally, screen readers help you show the relationship between text and form elements.

14.7.2 Aligning Form Elements with CSS

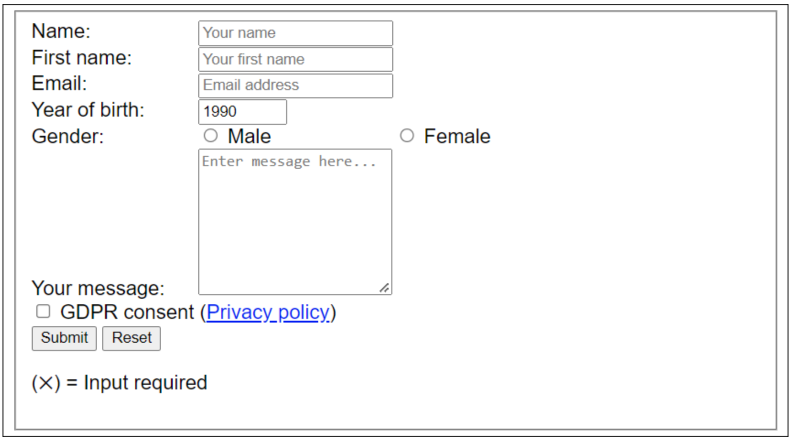
Once you've created the basic HTML framework with the relevant form elements, the first thing you should take care of is the alignment of the individual form elements. At the moment, everything is still a bit unordered (see Figure 14.58).

Just take a closer look at the figure: what you need first is a uniform alignment and a width for the labels on the left of the HTML form. You need to decide whether you want to display the labels next to the form elements or above them. In this example, I decided to place the labels next to the form elements, which I write in the following CSS snippet:

```
...
  label {
    min-width: 8em;
    display: inline-block;
    text-align: left;
  }
...
```

Listing 14.39 /examples/chapter014/14_7/css/style.css

Here, I've used `inline-block` for `display`, which allows you to treat an element like a block element and use the width and height properties as well as `margin` without creating a paragraph (as would be the case with `block`). If, on the other hand, you want to place the labels above the form elements for input, you should use `display: block;` instead.



Name: Your name

First name: Your first name

Email: Email address

Year of birth: 1990

Gender: ○ Male ○ Female

Enter message here...

Your message:

☐ GDPR consent (Privacy policy)

(X) = Input required

Figure 14.59 After the First Alignment of the HTML Element <label> with CSS

The first alignment of the `label` element looks good (see Figure 14.59). Next, you'll probably want to write a uniform width for the `input` input fields and the `textarea` field. Again, not much is needed to do that, as the following CSS lines show:

```
...
input {
  width: 20em;
  padding : 0.7em;
  font-family: Arial;
  color: gray;
}
textarea {
  width: 24em;
  padding: 0.1em;
  font-family: Arial;
  color: gray;
}
...
```

Listing 14.40 /examples/chapter014/14_7/css/style.css

Because the radio buttons and the checkbox are `input` input fields as well, and they would look strange with a width of `20em`, we use an extra selector for them. In the example, I've therefore written the following:

```
...
  input[type="checkbox"], input[type="radio"] {
    width: auto;
  }
...
```

Listing 14.41 /examples/chapter014/14_7/css/style.css

This only cancels the length assignment of the `input` fields of type `radio` and `checkbox`. Let's take a quick look at Figure 14.60—the interim result looks impressive, doesn't it?

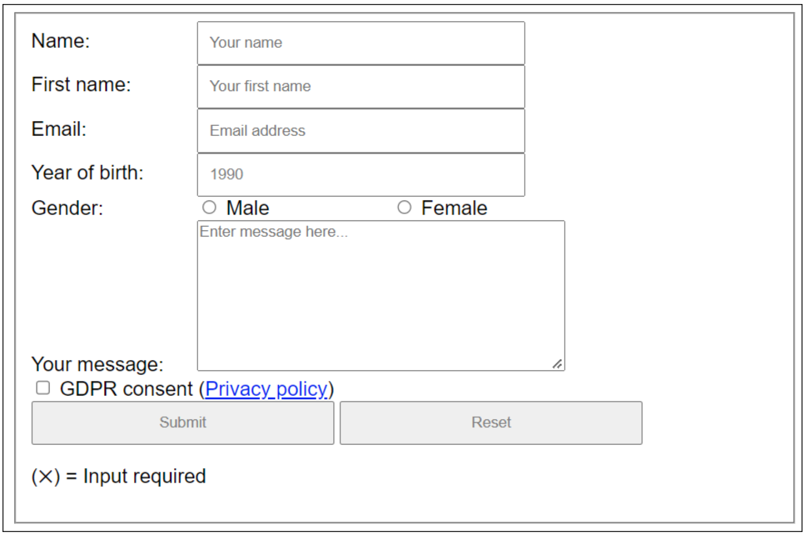

Name:
Your name
First name:
Your first name
Email:
Email address
Year of birth:
1990
Gender:
○ Male
○ Female
Enter message here...
Your message:
☐ GDPR consent (Privacy policy)
Submit
Reset
(X) = Input required

Figure 14.60 It's Starting to Look Neatly Arranged

What still doesn't look really appealing is the arrangement of the label in front of the multiline text field, the position of the checkbox below the text field, and the two buttons.

We should move the label in front of the multiline text field up as follows:

```
...
  label:first-child { vertical-align: top; }
...
```

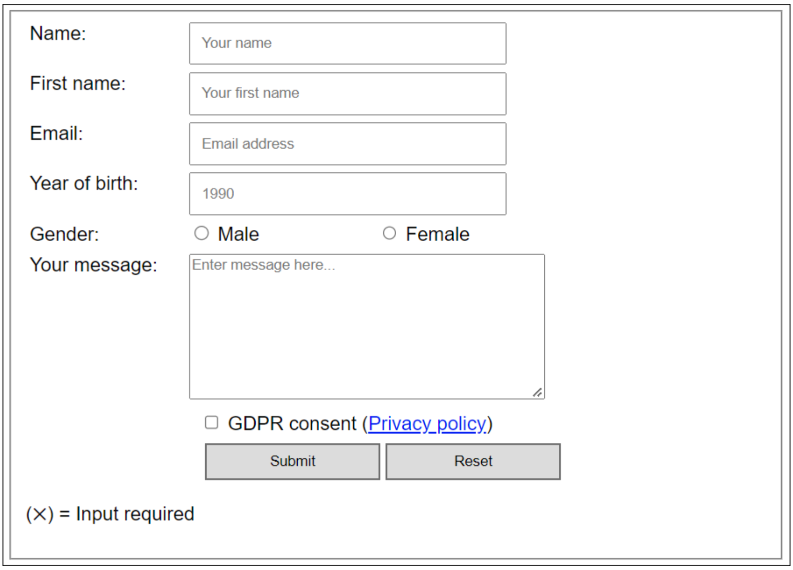
You can move the checkbox to the right by using a simple `margin-left`, and you can do the same with the `submit` button:

```
...
  input[type="checkbox"], input[type="submit"] {
    margin-left: 12em;
  }
...
```

Finally, a width specification for the two buttons is missing, which you can write as follows:

```
...
  input[type="submit"], input[type="reset"] {
    width: 12em;
  }
...
```

All in all, this completes the simple arrangement of the form elements with CSS. The result in Figure 14.61 looks very nice.



Name: Your name

First name: Your first name

Email: Email address

Year of birth: 1990

Gender: ○ Male ○ Female

Your message: Enter message here...

☐ GDPR consent (Privacy policy)

Submit Reset

(X) = Input required

Figure 14.61 Neatly Arranged Thanks to CSS

14.7.3 Designing Form Elements with CSS

After you've put everything in place, you can get down to styling the form visually with CSS. From the outside to the inside, the first thing to do is to design the `fieldset` and `legend`. In this example, I've used the following styles:

```
...
fieldset {
  width: 90%;
  padding-top: 1.5em;
  padding-left: 1.5em;
  background: rgb(240, 240, 240);
}

...
```

This way, we've styled the area between `<fieldset>` and `</fieldset>` by setting the width, a background color, and a border.

In addition to the visual design of form elements, you can also include interactions for visitors, which will make the website not only more beautiful, but easier to use. You can implement such interactions using the CSS pseudo-class `:hover` or `:active`. Here are some practical examples:

```
...
  input:hover, textarea:hover {
    background: #fffff0;
    border: 2px solid #efe816;
    box-shadow: 0 0 10px rgba(0,0,0,0.2);
  }
  input[type="submit"]:hover, input[type="reset"]:hover {
    background: #c9c9c9;
    border: 2px solid #6c6c6c;
  }
  input[type="submit"]:active, input[type="reset"]:active {
    background: #8f8f8f;
  }
...
```

If you now hover the mouse pointer over one of the `input` input fields or the multiline `textarea` input field (`:hover`), this input field will be displayed with a different color and a different border. The `box-shadow` feature also gives the impression that this input field is glowing.

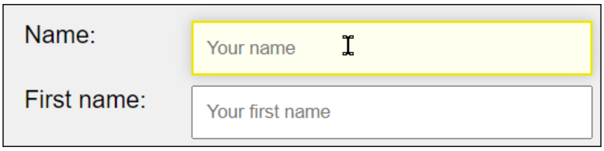


Figure 14.62 Interaction Help When the Mouse Pointer Is over an Input Field

The buttons in this example were also designed separately with CSS pseudo-classes, so that the background color changes when the mouse hovers over the button (`:hover`) or if you click on it (`:active`).

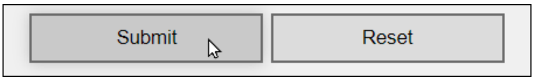
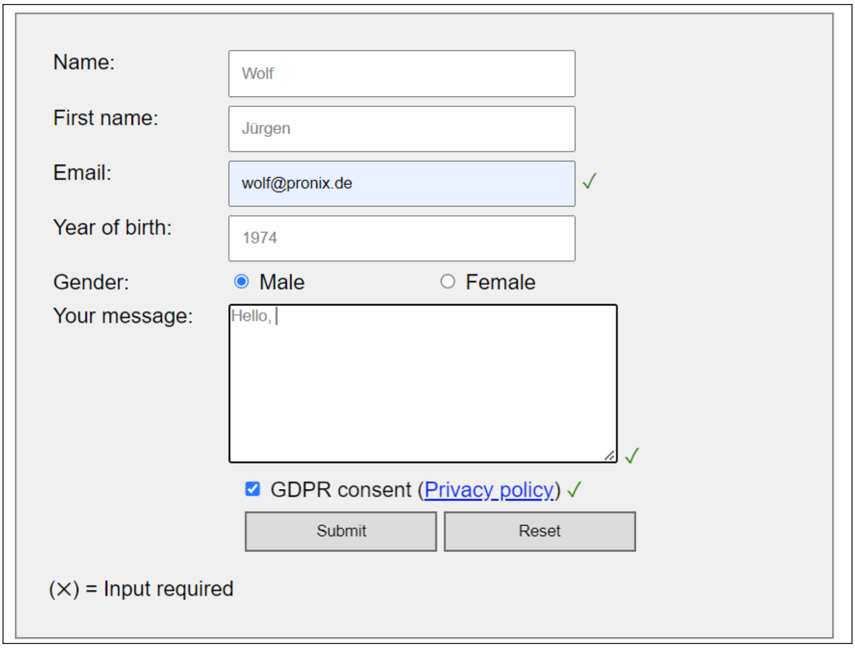


Figure 14.63 Hover Effect for Buttons with CSS

At the end of the example, I've created a version for smaller viewports. If the viewport width is less than 640 pixels, all `label` and `input` elements that aren't marked with the `.nobreak` class will be converted to block elements via `display:block;` and thus displayed one below the other. Here is the CSS for that:

```
...
/* Single column break at 640 pixels */
@media screen and (max-width: 40em) {
  label:not(.nobreak) { display: block; }
  label{ padding-bottom: 0.4em; }
  input:not(.nobreak) { display:block; }
  input[type="checkbox"], input[type="submit"], input[type="radio"]  {
    margin-left: 0;
  }
}
```

At this point, I could demonstrate and explain countless more options with CSS for HTML forms, but this would go beyond the scope of the book. The complete example for this section can be found in */examples/chapter014/14_7/index.html*.



Name: Wolf
First name: Jürgen
Email: wolf@pronix.de ✓
Year of birth: 1974
Gender: ● Male ○ Female
Your message: Hello, ✓
☑ GDPR consent (Privacy policy) ✓

(X) = Input required

Figure 14.64 A Simple HTML Form Styled with CSS

14.8 Summary

In this chapter, you've learned about many useful topics related to website styling. In detail, you should now be familiar with the following topics:

- How to style and customize ordinary text with CSS, as well as how to add web fonts using the `@font-face` rule
- How to style ordered and unordered lists with CSS, along with how to create a convenient navigation bar with lists
- How to style boring table data using CSS and make it much more attractive
- How to use special CSS features for tables
- How to deal with a few peculiarities of designing images and graphics with CSS
- How to transform elements using the CSS feature `transform`
- How to scale, rotate, skew, and move elements
- How to incorporate smoother transitions between transformation effects using the CSS feature `transition`
- How to properly align and style HTML forms using CSS

Chapter 15

Testing and Organizing

This chapter is more like a hodgepodge of different topics related to CSS and HTML. You'll learn specific things you still need to know in regards to testing and organizing websites.

In this chapter, you won't learn much about new features, but rather about techniques or tricks related to CSS and HTML that can be very useful. The following topics are described in this chapter:

- CSS and HTML are constantly evolving, and not every web browser can handle all new features right away. Here, you'll learn how to find out what a particular web browser is already capable of doing and what it isn't.
- Because there are more and more different devices and hence different screen sizes, here you'll learn how to test and view websites in different sizes using online tools.
- As projects get larger and longer, it can get confusing if you write everything in one CSS file. So, in this section, you'll learn how to use a central stylesheet to keep track of large projects.
- I'll also describe the built-in style defaults of a web browser and how to reset or normalize all CSS defaults.

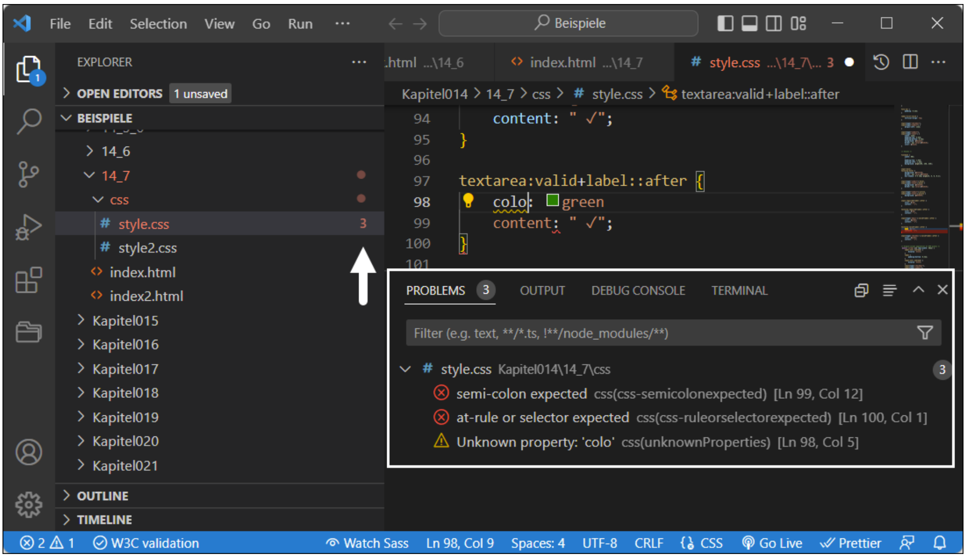
15.1 Web Browser Tests: What's Possible?

Today, most visitors are using modern web browsers. When this book went into print, Google Chrome was enthroned at the top, ahead of Safari. The remaining percentage points were shared between web browsers such as Firefox, Edge and Internet Explorer, Opera, and so on, although Firefox enjoys somewhat greater popularity in Germany than in the rest of the world. If you're serious about building websites, you'll be testing your work extensively in all major web browsers and on different devices. At this point, the question might arise which browsers you should use for testing and which functionality a web browser actually brings along. You'll get an answer to these questions in the following sections.

15.1.1 Validating HTML and CSS

The first step in testing a website should always involve validation. Many web browsers already provide a function for web developers to perform validation. Plug-ins are also often available for various browsers as well as HTML editors or development environments. Aside from that, the W3C with its online validator for HTML is available at *https://validator.w3.org* and for CSS at *https://jigsaw.w3.org/css-validator*.

At this point, it's important to mention that a valid website doesn't mean that it's also perfect at the same time. Things like accessibility, usability, or speed won't get better just because the HTML or CSS is valid. Consequently, those validators are also just another bunch of tools for quality assurance.


File
Edit
Selection
View
Go
Run
Beispiele
EXPLORER
OPEN EDITORS 1 unsaved
BEISPIELE
14_6
14_7
css
style.css 3
style2.css
index.html
index2.html
Kapitel015
Kapitel016
Kapitel017
Kapitel018
Kapitel019
Kapitel020
Kapitel021
OUTLINE
TIMELINE
.html ...\14_6
index.html ...\14_7
style.css ...\14_7\... 3
Kapitel014 > 14_7 > css > style.css > textarea:valid+label::after
PROBLEMS 3
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, **/*.ts, !**/node_modules/**)
⊗ 2 ⚠ 1
W3C validation
Watch Sass
Ln 98, Col 9
Spaces: 4
UTF-8
CRLF
CSS
Go Live
Prettier

```
    content: " √";
}

textarea:valid+label::after {
    colo: green
    content: " √";
}
```

style.css Kapitel014\14_7\css 3

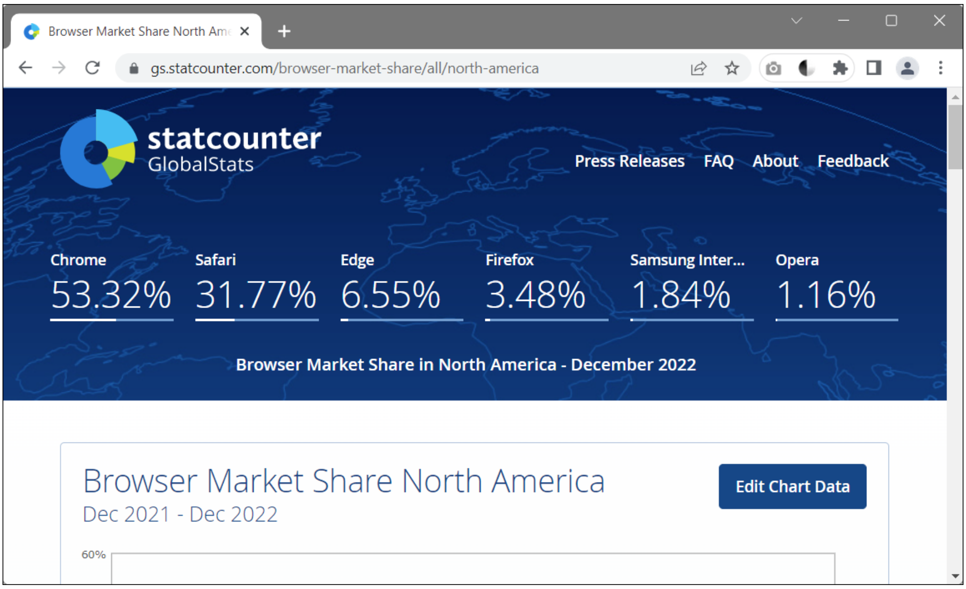
- semi-colon expected css(css-semicolonexpected) [Ln 99, Col 12]
- at-rule or selector expected css(css-ruleorselectorexpected) [Ln 100, Col 1]
- Unknown property: 'colo' css(unknownProperties) [Ln 98, Col 5]

Figure 15.1 What's Indispensable for Me Is a Validation of HTML and CSS during the Writing Process of HTML and CSS, Like Here with Visual Studio Code from Microsoft

15.1.2 Which Browsers Are Visitors Currently Using?

First of all, you might be wondering which browsers you should use for testing. Of course, it would be best to test with all browsers right away, but this already raises the question of feasibility, especially considering that most visitors are now using mobile devices. For this reason, it makes sense to first look at a few statistics about which browsers are most commonly used nowadays. You can find an interesting overview of this, for example, on the following websites: *www.w3counter.com/trends*, *http://gs.statcounter.com*, or *https://www.statista.com*.

Of course, you mustn't make the mistake of generalizing these kinds of statistics. For example, if you put a website online that contains mostly Apple articles, you're also likely to have mostly visitors with a Safari web browser. Then you'll probably study the statistics afterwards anyway, when the website has gone online, to see what your visitors came to your website with.



Browser Market Share North Am
gs.statcounter.com/browser-market-share/all/north-america
statcounter
GlobalStats
Press Releases
FAQ
About
Feedback
Edit Chart Data
60%

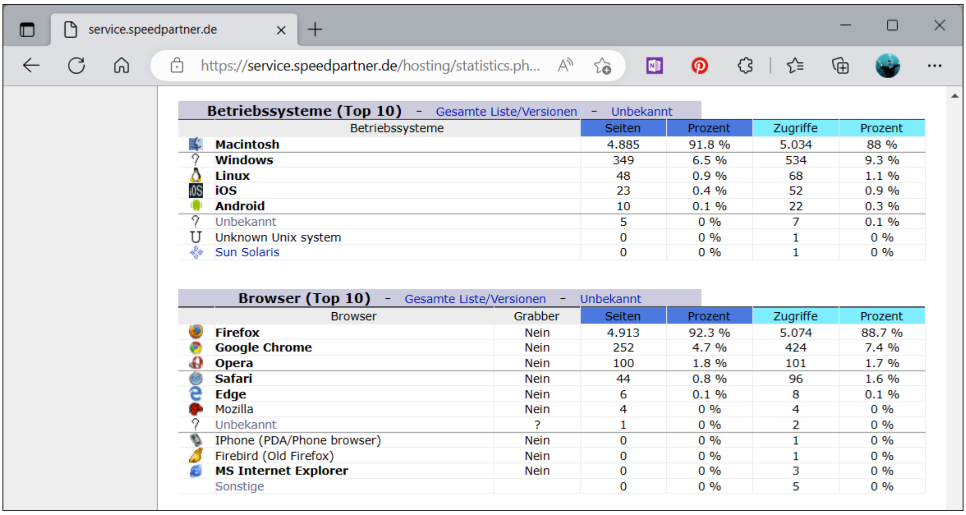
Chrome	Safari	Edge	Firefox	Samsung Inter...	Opera
53.32%	31.77%	6.55%	3.48%	1.84%	1.16%

Browser Market Share in North America - December 2022

Browser Market Share North America

Dec 2021 - Dec 2022

Figure 15.2 Web Browser Market in Germany



service.speedpartner.de
https://service.speedpartner.de/hosting/statistics.ph...

Betriebssysteme (Top 10) - Gesamte Liste/Versionen - Unbekannt

Betriebssysteme	Seiten	Prozent	Zugriffe	Prozent
Macintosh	4.885	91.8 %	5.034	88 %
Windows	349	6.5 %	534	9.3 %
Linux	48	0.9 %	68	1.1 %
iOS	23	0.4 %	52	0.9 %
Android	10	0.1 %	22	0.3 %
Unbekannt	5	0 %	7	0.1 %
Unknown Unix system	0	0 %	1	0 %
Sun Solaris	0	0 %	1	0 %

Browser (Top 10) - Gesamte Liste/Versionen - Unbekannt

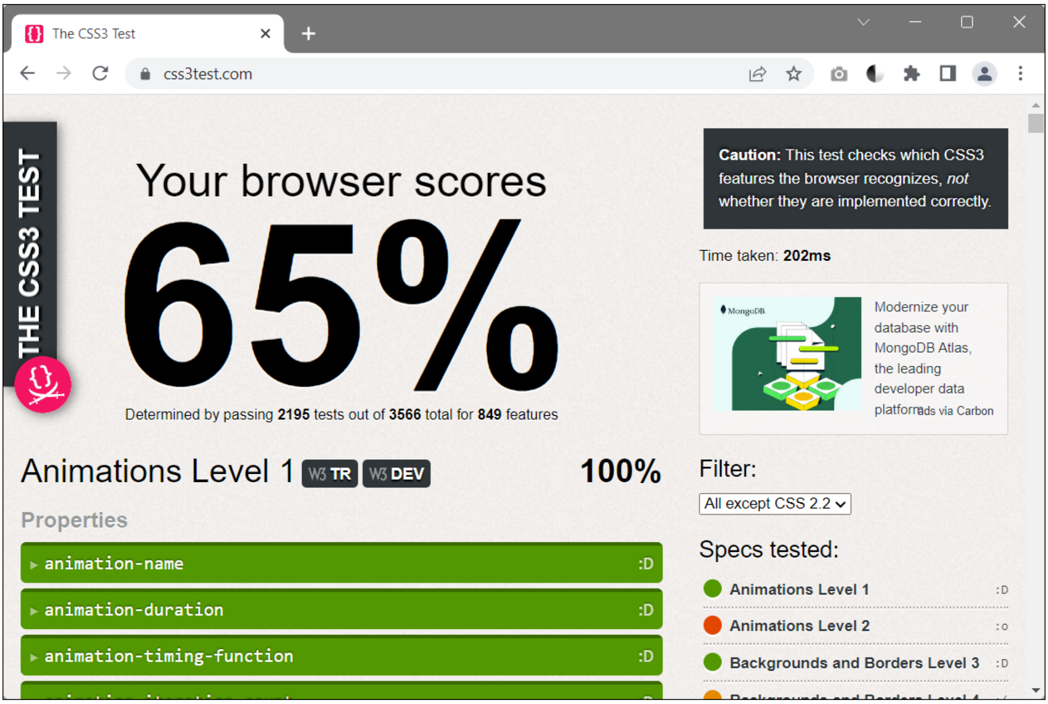
Browser	Grabber	Seiten	Prozent	Zugriffe	Prozent
Firefox	Nein	4.913	92.3 %	5.074	88.7 %
Google Chrome	Nein	252	4.7 %	424	7.4 %
Opera	Nein	100	1.8 %	101	1.7 %
Safari	Nein	44	0.8 %	96	1.6 %
Edge	Nein	6	0.1 %	8	0.1 %
Mozilla	Nein	4	0 %	4	0 %
Unbekannt	?	1	0 %	2	0 %
IPhone (PDA/Phone browser)	Nein	0	0 %	1	0 %
Firebird (Old Firefox)	Nein	0	0 %	1	0 %
MS Internet Explorer	Nein	0	0 %	3	0 %
Sonstige		0	0 %	5	0 %

Figure 15.3 A Look at Your Own Statistics Then Reveals More Precisely What Your Visitors Really Use to Visit the Website

15.1.3 CSS Web Browser Test

Fortunately, modern CSS support is very good in all modern web browsers. To test the capabilities of the web browser with regard to CSS, there are several test systems available on the web. The advantage of those test systems should be that you can at least weigh up during the development of your website whether you should use a new CSS feature on your website at all or set up a fallback for certain web browsers that can’t handle it.

A more specific test for CSS features was developed by Lea Verou. You can perform this test online by entering the address *http://css3test.com* in your web browser. The latest web browsers currently manage around 50% to 67%, which isn’t bad at all. It’s also useful that the test lists the results for each individual area as well. Nice side effect of this test: you’ll discover many new CSS features and also immediate links to the corresponding entries of the specification.



Your browser scores
65%
Determined by passing **2195** tests out of **3566** total for **849** features

Caution: This test checks which CSS3 features the browser recognizes, *not* whether they are implemented correctly.

Time taken: **202ms**

Modernize your database with MongoDB Atlas, the leading developer data platform
ads via Carbon

Animations Level 1 W3 TR W3 DEV **100%**

Properties

- animation-name :D
- animation-duration :D
- animation-timing-function :D

Filter:
All except CSS 2.2

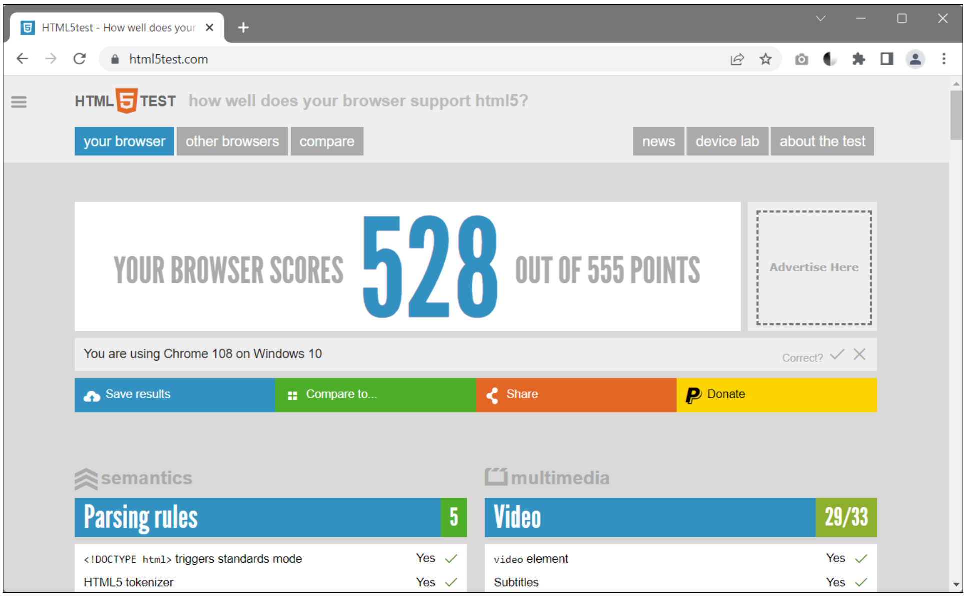
Specs tested:

- ● Animations Level 1 :D
- ● Animations Level 2 :o
- ● Backgrounds and Borders Level 3 :D

Figure 15.4 On https://css3test.com, You Get a Nice List of What the Web Browser Can and Can’t (Yet) Do in Detail

15.1.4 HTML5 Web Browser Test

As for CSS, you can find an interesting overview of what the web browser can do and what it can’t do (yet) compared to other web browsers in terms of HTML at *https://html5test.com*. Here, too, you’ll find the latest HTML features linked to the corresponding specification.


HTML5test - How well does your
html5test.com
HTML5 TEST
how well does your browser support html5?
your browser
other browsers
compare
news
device lab
about the test
Advertise Here
Correct?
Save results
Compare to...
Share
Donate
semantics
multimedia

YOUR BROWSER SCORES 528 OUT OF 555 POINTS

You are using Chrome 108 on Windows 10

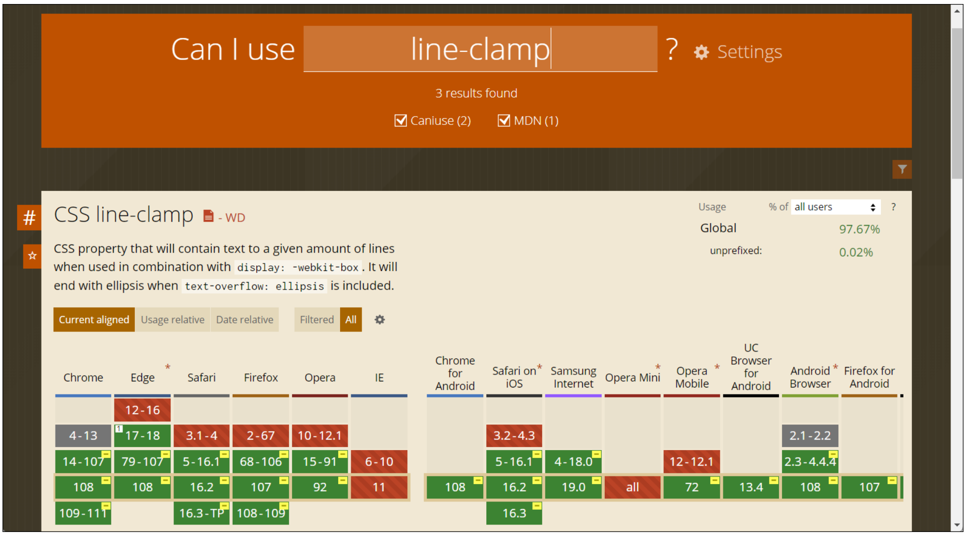
Parsing rules	5
`<!DOCTYPE html>` triggers standards mode	Yes ✓
HTML5 tokenizer	Yes ✓

Video	29/33
`video` element	Yes ✓
Subtitles	Yes ✓

Figure 15.5 What the Web Browser Do in Terms of HTML

15.1.5 Can I Use That?

It's not easy to keep track of the different web technologies and what you can use of them with which web browser. Especially with newer CSS and HTML features, it can become quite a tedious matter to test what already works on which web browser without web browser prefixes.


Can I use
line-clamp
?
Settings
3 results found
☑ Caniuse (2)
☑ MDN (1)
#
☆
- WD
Usage
% of
all users
?
Current aligned
Usage relative
Date relative
Filtered
All

CSS line-clamp

CSS property that will contain text to a given amount of lines when used in combination with `display: -webkit-box`. It will end with ellipsis when `text-overflow: ellipsis` is included.

Global 97.67%
unprefixed: 0.02%

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *	UC Browser for Android	Android Browser *	Firefox for Android
	12-16												
4-13	17-18	3.1-4	2-67	10-12.1			3.2-4.3					2.1-2.2	
14-107	79-107	5-16.1	68-106	15-91	6-10		5-16.1	4-18.0		12-12.1		2.3-4.4.4	
108	108	16.2	107	92	11	108	16.2	19.0	all	72	13.4	108	107
109-111		16.3-TP	108-109				16.3						

Figure 15.6 The Web Database www.caniuse.com Is Very Useful When It Comes to Determining Which Web Techniques Can Be Used with Which Web Browser

The web database at *www.caniuse.com* specializes in such cases and has proven its worth. The database takes into account the currently popular web browsers.

The website also features a very helpful continuing list of issues, notes, and resources on the topic you're looking for. Searching for a topic is comfortable and easy thanks to a dynamic search function.

15.1.6 Feature Query Using the "@supports" Rule

You can check whether a particular CSS feature is supported by a web browser using the CSS rule `@supports`. This feature query is also referred to as a *CSS feature query*. Here's an example:

```
...
@supports( hyphens: auto) {
  p {
    text-align: justify;
    hyphens: auto;
  }
}
...
```

Listing 15.1 /examples/chapter015/15_1_6/css/style.css

Here, we use `@supports` to check if the web browser understands the `hyphens: auto` feature. If that's the case, the text in the `p` elements will be `justified`, and the CSS feature `hyphens` will be set to `auto`. You can use the CSS feature `hyphens` to enable the automatic `hyphenation` function of the web browser.

As an alternative, you can use the `@supports` rule with the `not` operator to check if the web browser doesn't support a specific CSS feature. For example:

```
@supports not( display: grid ) {
  /* CSS features in case the browser does not know display: grid */
  float: right; /* e.g. float: right as alternative */
}
```

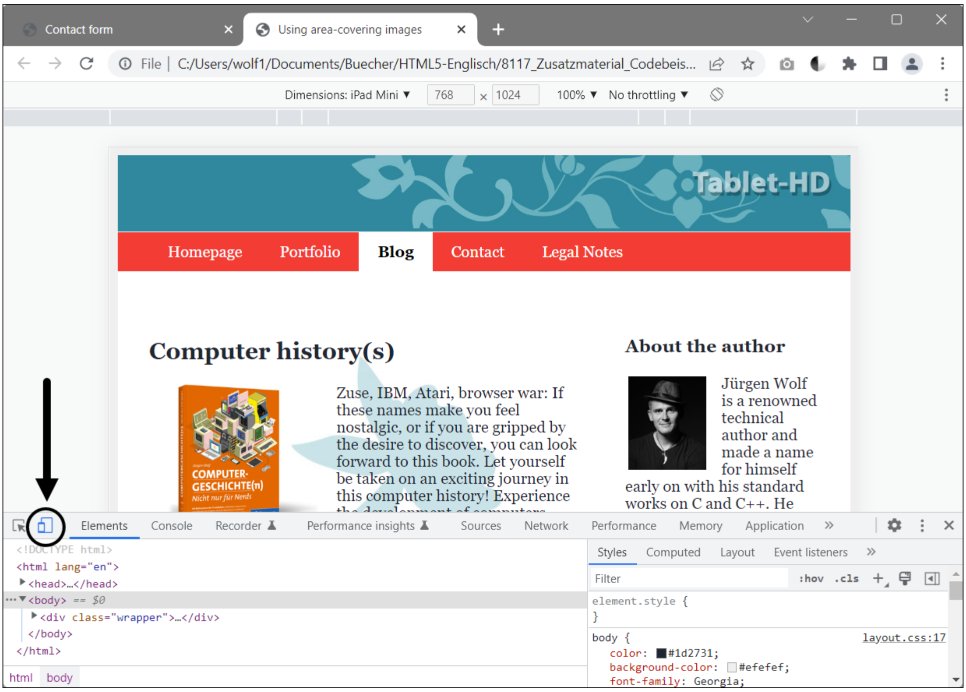
You can also combine multiple CSS features by using `and` and `or`. Moreover, `@media` and `@supports` rules can be nested. Web browsers that don't understand the `@supports` rule will ignore anything inside the curly brackets.

15.2 Viewing Websites in Different Sizes

In addition to testing websites in different web browsers and validating CSS and HTML code, you should also view the website on screens of different sizes. If you've developed a website that responds to viewport width with media queries, you can also track a layout break to the next smaller or next larger viewport using the desktop browser by manually changing the browser width. In practice, however, this is somewhat inconvenient and imprecise in the long run.

For viewing websites in different sizes, almost all web browser manufacturers now offer their own tools for viewing websites in different sizes, some of which are integrated into the web browser:

- **Firefox**
  In Firefox, for example, you can find the **Test Screen Sizes** command in the **Tools • Web Developer** menu or use the keyboard shortcut `Ctrl` + `Shift` + `M` (Windows) or `Alt` + `cmd` + `M` (Mac), which also allows you to view the current web page in different screen sizes.
- **Chrome, Edge**
  You can also find a corresponding tool in Chrome and Edge via the menu **Display • More tools • Developer tools**. Again, the keyboard shortcut `Ctrl` + `Shift` + `M` (Windows) or `Alt` + `cmd` + `M` (Mac) will get you there faster.
- **Safari**
  In Safari, you must first enable the **Developer** menu via **Safari • Preferences** in the **Advanced** tab. Then, you'll find the item **Switch to "Responsive Design" mode** in the **Developer** menu. The shortcut `Alt` + `cmd` + `R` will also get you there.


Contact form
Using area-covering images
File | C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeis...
Dimensions: iPad Mini ▼
768
1024
100% ▼
No throttling ▼
Tablet-HD
Homepage
Portfolio
Blog
Contact
Legal Notes
COMPUTER-GESCHICHTE(n)
Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory
Application
Styles
Computed
Layout
Event listeners
Filter
:hov
.cls
html
body

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience

About the author

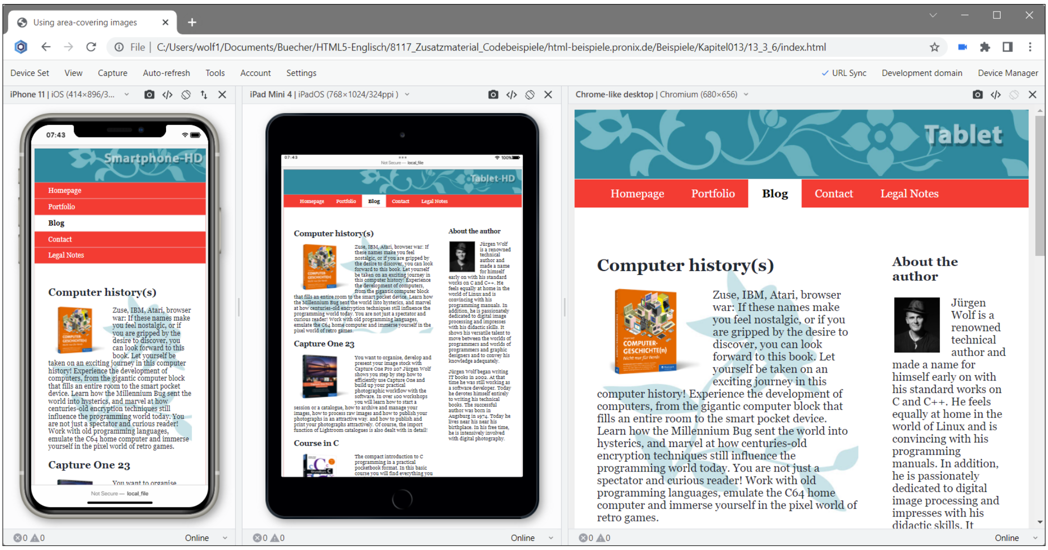
Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
…▼<body> == $0
   ▶<div class="wrapper">…</div>
  </body>
</html>
```

```
element.style {
}
body {                                  layout.css:17
   color: ■#1d2731;
   background-color: □#efefef;
   font-family: Georgia;
```

Figure 15.7 Testing Screen Sizes Using Google Chrome

My personal favorite tool to test a website on different devices and with different screen resolutions is the commercial web browser Blisk (*https://blisk.io*), which is completely based on Chromium. This browser includes tools that make testing desktop and mobile versions during development even more efficient.


Using area-covering images
File | C:/Users/wolf1/Documents/Buecher/HTML5-Englisch/8117_Zusatzmaterial_Codebeispiele/html-beispiele.pronix.de/Beispiele/Kapitel013/13_3_6/index.html
Device Set
View
Capture
Auto-refresh
Tools
Account
Settings
URL Sync
Development domain
Device Manager
iPhone 11 | iOS (414×896/3...
iPad Mini 4 | iPadOS (768×1024/324ppi)
Chrome-like desktop | Chromium (680×656)
Smartphone-HD
Tablet-HD
Tablet
Homepage
Portfolio
Blog
Contact
Legal Notes
Online

Computer history(s)

Zuse, IBM, Atari, browser war: If these names make you feel nostalgic, or if you are gripped by the desire to discover, you can look forward to this book. Let yourself be taken on an exciting journey in this computer history! Experience the development of computers, from the gigantic computer block that fills an entire room to the smart pocket device. Learn how the Millennium Bug sent the world into hysterics, and marvel at how centuries-old encryption techniques still influence the programming world today. You are not just a spectator and curious reader! Work with old programming languages, emulate the C64 home computer and immerse yourself in the pixel world of retro games.

About the author

Jürgen Wolf is a renowned technical author and made a name for himself early on with his standard works on C and C++. He feels equally at home in the world of Linux and is convincing with his programming manuals. In addition, he is passionately dedicated to digital image processing and impresses with his didactic skills. It

Figure 15.8 The Blisk Web Browser Allows You to Test a Website on Different Devices and Screen Sizes

15.3 Setting Up a Central Stylesheet

You may have seen CSS files on the web that you thought were pretty large. Especially if you're actively developing a more extensive website, it can be useful to *initially* distribute the various style statements across multiple CSS files.

The principle is simple: You use a central stylesheet, which you include in the HTML document as usual via the `link` element. In Figure 15.9, *style.css* is the central stylesheet. However, this central stylesheet doesn't contain any ordinary CSS content, but again only loads the other CSS files (e.g., *reset.css*, *print.css*, *layout.css*, *navi.css*, and *iebrowser.css* in the example of the figure) by using the `@import` rule. At first, this may sound a bit cumbersome, but during the development of extensive web projects, it's rather a relief to divide the stylesheets into meaningful units such as layout of header and footer, navigation, content, layout for printing, layout for old web browsers and so on—here you must decide by yourself according to sense and personal taste how (and if) you want to divide the stylesheets.

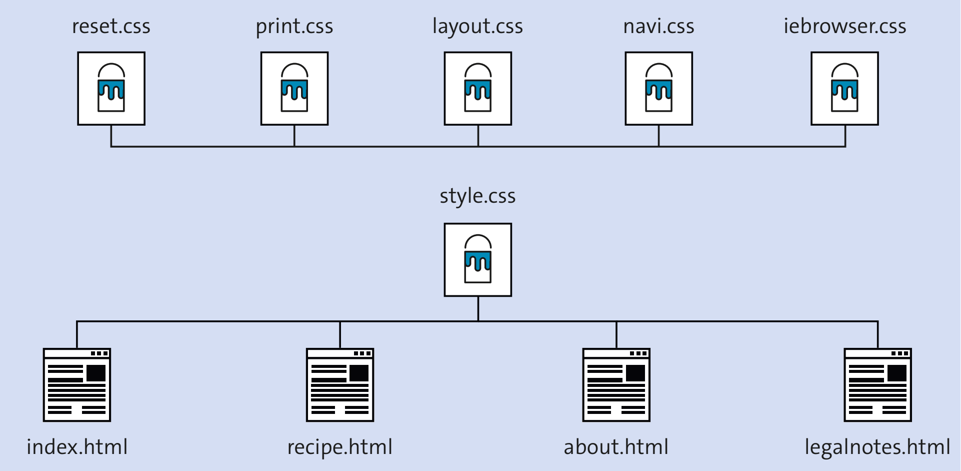

reset.css
print.css
layout.css
navi.css
iebrowser.css
style.css
index.html
recipe.html
about.html
legalnotes.html

Figure 15.9 A Central Stylesheet Helps to Keep an Overview during Development and to Track Down Errors More Quickly

With reference to the example in Figure 15.9, such a central stylesheet *style.css* could look like the following:

```
/* Example of a central stylesheet, style.css */

/* Instead of reset.css, normalize.css would also work,
   which often makes more sense than reset.css.
   For more information, see Section 15.4.2 */

@import url("reset.css");

/* Basic design */
@import url("layout.css");

/* Navigation */
@import url("navi.css");

/* Print version */
@import url("print.css");
...
```

CSS Files in a Separate Folder

In practice, it's also recommended to store CSS files generally in a separate directory. For this reason, you can often find the directory name *CSS*.

15.3.1 Combining Everything Back into One File to Shorten the Load Time

Keep in mind that the emphasis was on *development*. The disadvantage of multiple individual CSS files is that multiple server requests (one for each file) become necessary, which of course can increase the load time of the website significantly. So, when you're done with the website and want to make it public, you should combine the individual CSS files into one file for speed reasons. As in Figure 15.9, you can leave it at *style.css* because this file was included in the HTML document anyway.

Usually, you need to copy the style statements of each CSS file (as in Figure 15.9 with *reset.css*, *print.css*, *layout.css*, and *navi.css*) to the clipboard and paste them into *style.css*. However, if you use CSS statements for old web browsers, as in the *iebrowser.css* example, they should still be provided in a separate file.

If you want to avoid this kind of effort in the future and automate the development process a little more, you should take a look at the development tools *Grunt* (*http://gruntjs.com*) or *Gulp* (*http://gulpjs.com*).

CSS Compression

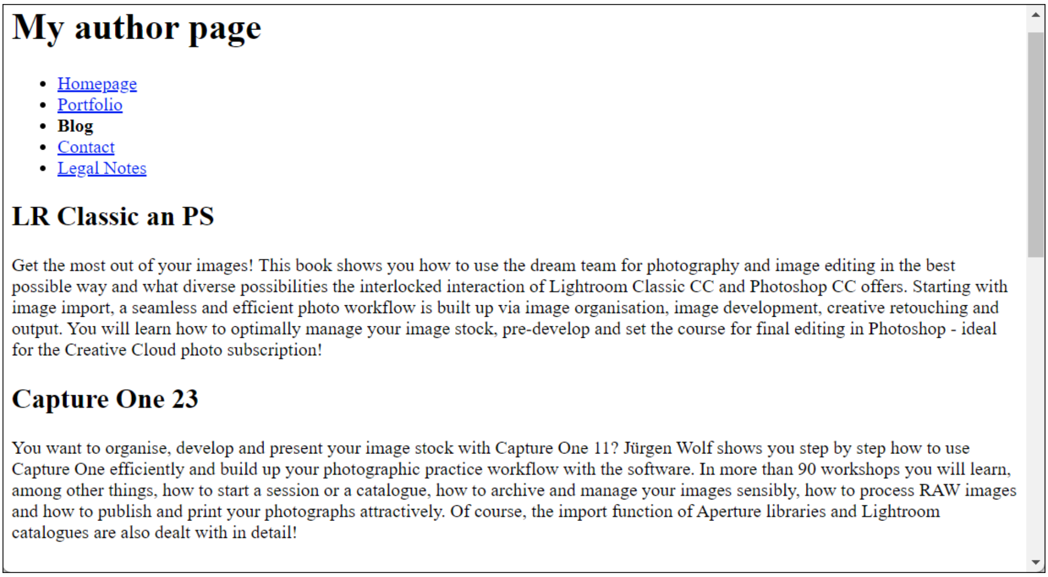
To reduce the file size of the CSS file again a bit and thus improve the load time, you could remove all superfluous lines of code with white spaces, line breaks, and comments. For such tasks, there are online tools available such as *CSS Compressor* (*https://cssminifier.com*) or *YUI Compressor* (*http://refresh-sf.com*). Before you do that, you should make a backup copy of your CSS file, which you'll need again if you want to change anything. The CSS file is no longer pleasant to read and edit after a CSS compression.

15.4 CSS Reset or Normalization?

The goal and purpose of a reset or normalization is to put the different basic browser settings on a common basis as much as possible, so that the website contains as few differences as possible in the different web browsers. To create such a CSS base, two ways have been established: reset and normalization.

15.4.1 Built-In Style Presets of the Web Browser and CSS Reset

When viewing the HTML document in different web browsers, you may have noticed that the display differs slightly. This is because all web browsers have built-in stylesheet defaults. One option would be to override the defaults with a *CSS reset*, so that when you start designing stylesheets, you're virtually doing everything yourself from the start.



My author page

- Homepage
- Portfolio
- **Blog**
- Contact
- Legal Notes

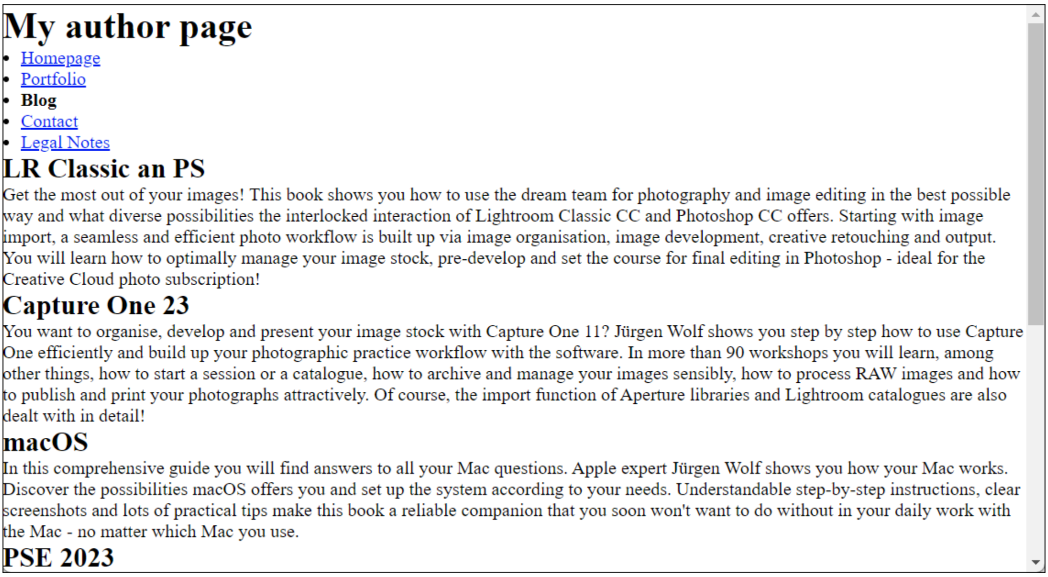
LR Classic an PS

Get the most out of your images! This book shows you how to use the dream team for photography and image editing in the best possible way and what diverse possibilities the interlocked interaction of Lightroom Classic CC and Photoshop CC offers. Starting with image import, a seamless and efficient photo workflow is built up via image organisation, image development, creative retouching and output. You will learn how to optimally manage your image stock, pre-develop and set the course for final editing in Photoshop - ideal for the Creative Cloud photo subscription!

Capture One 23

You want to organise, develop and present your image stock with Capture One 11? Jürgen Wolf shows you step by step how to use Capture One efficiently and build up your photographic practice workflow with the software. In more than 90 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to process RAW images and how to publish and print your photographs attractively. Of course, the import function of Aperture libraries and Lightroom catalogues are also dealt with in detail!

Figure 15.10 For Example, This Is What the Built-In Stylesheet Looks Like in the Chrome Web Browser



My author page

- Homepage
- Portfolio
- **Blog**
- Contact
- Legal Notes

LR Classic an PS

Get the most out of your images! This book shows you how to use the dream team for photography and image editing in the best possible way and what diverse possibilities the interlocked interaction of Lightroom Classic CC and Photoshop CC offers. Starting with image import, a seamless and efficient photo workflow is built up via image organisation, image development, creative retouching and output. You will learn how to optimally manage your image stock, pre-develop and set the course for final editing in Photoshop - ideal for the Creative Cloud photo subscription!

Capture One 23

You want to organise, develop and present your image stock with Capture One 11? Jürgen Wolf shows you step by step how to use Capture One efficiently and build up your photographic practice workflow with the software. In more than 90 workshops you will learn, among other things, how to start a session or a catalogue, how to archive and manage your images sensibly, how to process RAW images and how to publish and print your photographs attractively. Of course, the import function of Aperture libraries and Lightroom catalogues are also dealt with in detail!

macOS

In this comprehensive guide you will find answers to all your Mac questions. Apple expert Jürgen Wolf shows you how your Mac works. Discover the possibilities macOS offers you and set up the system according to your needs. Understandable step-by-step instructions, clear screenshots and lots of practical tips make this book a reliable companion that you soon won't want to do without in your daily work with the Mac - no matter which Mac you use.

PSE 2023

Figure 15.11 This Is What the Built-In Stylesheet Looks Like When You Use a CSS Reset to Override the Built-In Stylesheet

In Figure 15.11, the built-in stylesheet of the web browser was reset using the * selector (= universal selector) with the following CSS statements:

```
* {
    margin: 0;
    padding: 0;
}
```

This removes all external and internal spacings and borders on all elements. In practice, this will have to be done at many points in a project anyway. Such a CSS reset is often a bit too radical because here even the lists are without indentation, and you have to set spacing again yourself for every smallest detail. However, because of the lists, it wouldn't be too much effort if you summarized the CSS reset as follows:

```
...
/* Minimal version of a CSS reset */
* {
    margin: 0;
    padding: 0;
}
ul, ol {
    margin-left: 1em;
}
...
```

Listing 15.2 /examples/chapter015/15_4/css/reset.css

A much better and ready solution for a reset stylesheet by Eric Meyer can be found at *http://meyerweb.com/eric/tools/css/reset/*. You can copy and paste this CSS code to your project and modify it if necessary.

15.4.2 Normalization: The Alternative to CSS Reset

The gentler and arguably better alternative to the hard CSS reset of the web browser's built-in stylesheet defaults is *normalization*. Although normalization overrides many built-in stylesheet defaults of the web browser, it does so while respecting *useful* CSS defaults. Fortunately, you don't have to start by thinking about what constitutes useful defaults yourself because others have already done that for you. You can find multiple normalization projects on the internet, which you can download and include in your project. And if you don't like some of the defaults, you can and should modify them and adapt them to your own needs.

Probably the most famous normalization project is *normalize.css* by Nicolas Gallagher, which you can download from the website, *https://necolas.github.io/normalize.css/*. What's particularly nice about this project is that it fixes several web browser bugs right away.

The Evolution of Normalization: “sanitize.css”

From the normalization project with *normalize.css*, the *sanitize.css* project by Jonathan Neal has emerged. Unlike *normalize.css*, the project no longer takes into account older web browsers such as Safari 8 or Internet Explorer 10 and instead introduces constructs from newer web browsers. The *sanitize.css* project can be found at *https://css-tools.github.io/sanitize.css/*.

CSS Reset, Normalization, or Leave Everything as It Is?

At this point, as with the central stylesheet setup, I should mention that these are all mere suggestions. Depending on their preferences, some web designers do a hard CSS reset and take over all tasks themselves, leaving nothing to chance or default settings, whereas others use one of the normalization projects and adapt the template to their own needs. Still others aren’t fond of a CSS reset or normalization at all and prefer to work with their own styles.

There has been discussion about the need for a CSS reset for some time. Many web developers question the sense of first defining a complete CSS reset and then immediately overwriting it with their own styles. You’ll come across a lot of pros and cons on the web and will probably have to ultimately decide for yourself what works best for you personally.

15.5 Summary

In this chapter, you’ve learned some useful things that can make your life with CSS easier. If you’ve worked through the chapter, you now know the following:

- How to test a browser to see what it can and can’t do in terms of CSS
- How to view and test websites in different screen sizes with one tool
- How to organize large web projects more clearly with a central stylesheet
- How to completely reset or normalize the web browser defaults by means of a CSS reset

Chapter 16

The CSS Preprocessor Sass and SCSS

A chapter on CSS preprocessors is almost mandatory in a book on CSS these days. You can use a CSS preprocessor to make writing CSS easier, for example, by eliminating repetitive writing. Code handling can also be simplified with a CSS preprocessor. This chapter contains a brief introduction to this topic; for this purpose, I chose Sass as the CSS preprocessor.

It may seem a bit inconvenient at first to pull another technology on top of CSS, which ultimately generates just another CSS file. But we're talking about CSS preprocessors. Sass is just one of many other CSS preprocessors that are now being used briskly by serious web designers on a daily basis. A preprocessor is used to automate tedious tasks and provide new functionality. A simple example is if, when creating a website with CSS, you assign a red color countless times to various CSS properties because that's the color a customer wants. However, the customer changes their mind and would rather have a shade of blue, so you have to assign the blue color to all red elements. This is where a CSS preprocessor such as Sass comes into play. Instead of constantly changing repetitive values, you can implement the change at only one central point according to the DRY principle (don't repeat yourself).

Other well-known representatives of CSS preprocessors are, for example, Less or Stylus. The special case or difference of Sass and SCSS will be explained in the following sections of this chapter. If you compare the CSS preprocessors with each other, you'll find that many things are pretty similar. I chose Sass because I use it a lot myself, and Sass is probably the CSS preprocessor with the biggest community.

The goal of this chapter is to familiarize you with the basics of Sass. The examples are therefore usually very simple and shorter in nature. But the advantage of using a CSS preprocessor to develop CSS code may not be immediately apparent to you. However, this will change significantly when you use Sass in more extensive projects.

16.1 Sass or SCSS Syntax

Because Sass and SCSS are often mentioned at the same time, it's beginners in particular who wonder whether these are two different CSS preprocessors. The answer is yes and no because both are *Syntactically Awesome Style Sheet* (Sass) after all. The difference between the two is that they're two different grammars. The original syntax was *Sass syntax*, and the syntax that was introduced afterward was *Sassy CSS* (SCSS). In this book, I use the newer SCSS syntax, which uses curly brackets and semicolons, unlike the Sass syntax. The following example in Table 16.1 is intended to briefly explain the difference between the two syntaxes without going into too much detail.

SCSS Syntax (style.scss)

```
$but-size: 100%;
$color1: #fcfcfc;
$color2: #fafafa;
$spacing-p: 1em;
$spacing-m: 0.5em;

.button-form {
  width: $but-size;
  padding: $spacing-p;
  margin: $spacing-m;
  background-color: $color1;
  &:hover {
  background-color: $color2;
  color: $color1;
  }
}
```

Sass Syntax (style.sass)

```
$but-size: 100%
$color1: #fcfcfc
$color2: #fafafa
$spacing-p: 1em
$spacing-m: 0.5em

.button-form
  width: $but-size
  padding: $spacing-p
  margin: $spacing-m
  background-color: $color1
  &:hover
  background-color: $color2
  color: $color1
```

Table 16.1 Differences between SCSS Syntax and Sass Syntax

The Sass syntax is always a bit shorter because, as already mentioned, it doesn't use curly brackets and semicolons. In Sass, nesting is done by means of indentation.

16.2 From Sass/SCSS to CSS

Of course, for styling purposes, the web browser can't do anything with the SCSS file shown on the left in Table 16.2. A file with the SCSS syntax has the extension **.scss* (for the Sass syntax, it's **.sass*). For this purpose, the CSS preprocessor must first convert (compile) the SCSS file into a CSS file, which you then also use for the web browser. On the right-hand side in Table 16.2, you can see the result of the SCSS file after the CSS preprocessor run as a CSS file.

SCSS File (style.scss)

```
$but-size: 100%;
$color1: #fcfcfc;
```

CSS File (style.css)

```
.button-form {
  width: 100%;
```

Table 16.2 From an SCSS File to a CSS File after the CSS Preprocessor Run

SCSS File (style.scss)	CSS File (style.css)

SCSS File (style.scss)

```
$color2: #fafafa;
$spacing-p: 1em;
$spacing-m: 0.5em;

.button-form {
  width: $but-size;
  padding: $spacing-p;
  margin: $spacing-m;
  background-color: $color1;
  &:hover {
  background-color: $color2;
  color: $color1;
  }
}
```

CSS File (style.css)

```
  padding: 1em;
  margin: 0.5em;
  background-color: #fcfcfc;
}

.button-form:hover {
  background-color: #fafafa;
  color: #fcfcfc;
}
```

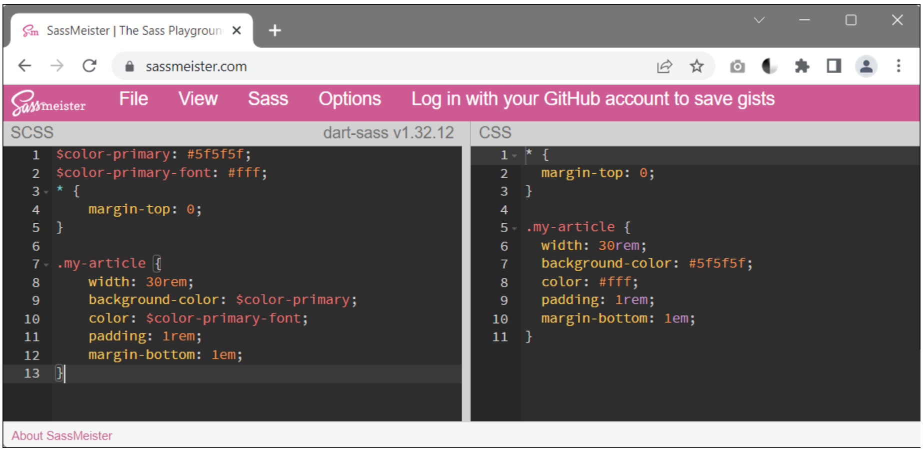
Table 16.2 From an SCSS File to a CSS File after the CSS Preprocessor Run (Cont.)

16.3 Installing and Setting Up Sass

Now that you know you need a CSS preprocessor that compiles a CSS file from the SCSS file, let's take a brief look at some possible options.

16.3.1 Online CSS Preprocessor without Installation

If you want to get started right away and aren't yet sure if you want to use Sass, or don't feel like installing and setting up an environment, you can take a look at the Sassmeister website (*www.sassmeister.com*). There you'll find an online compiler that compiles an SCSS syntax (and also the Sass syntax) to CSS.


SassMeister | The Sass Playgroun
sassmeister.com
Sassmeister
File
View
Sass
Options
Log in with your GitHub account to save gists
SCSS
dart-sass v1.32.12
CSS
About SassMeister

```
$color-primary: #5f5f5f;
$color-primary-font: #fff;
* {
    margin-top: 0;
}

.my-article {
    width: 30rem;
    background-color: $color-primary;
    color: $color-primary-font;
    padding: 1rem;
    margin-bottom: 1em;
}
```

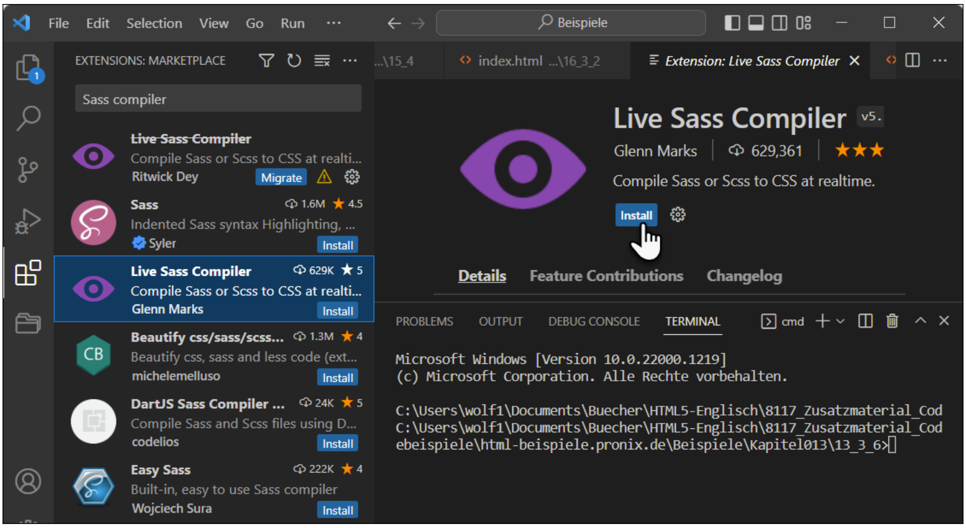
```
* {
  margin-top: 0;
}

.my-article {
  width: 30rem;
  background-color: #5f5f5f;
  color: #fff;
  padding: 1rem;
  margin-bottom: 1em;
}
```

Figure 16.1 The Sassmeister Website Provides an Online CSS Preprocessor

16.3.2 Setting Up Sass Using Visual Studio Code

My favorite solution is to set up Sass with Visual Studio Code. First, it can be done with a few clicks, and second, the Visual Studio development environment is available on all common systems and enjoys great popularity.

To install Sass with Visual Studio Code, you want to click on the icon with **Extensions** on the left. In the search bar, enter “Sass compiler”, and select **Live Sass Compiler**. Then click **Install** in the description window. Restart Visual Studio Code.


File
Edit
Selection
View
Go
Run
Beispiele
EXTENSIONS: MARKETPLACE
Sass compiler
~~Live Sass Compiler~~
Compile Sass or Scss to CSS at realti...
Ritwick Dey
Migrate
Sass
1.6M
4.5
Indented Sass syntax Highlighting, ...
Syler
Install
Live Sass Compiler
629K
5
Compile Sass or Scss to CSS at realti...
Glenn Marks
Install
Beautify css/sass/scss...
1.3M
4
Beautify css, sass and less code (ext...
michelemelluso
Install
DartJS Sass Compiler ...
24K
5
Compile Sass and Scss files using D...
codelios
Install
Easy Sass
222K
4
Built-in, easy to use Sass compiler
Wojciech Sura
Install
...\15_4
index.html ...\16_3_2
Extension: Live Sass Compiler
Live Sass Compiler
v5.
Glenn Marks
629,361
Compile Sass or Scss to CSS at realtime.
Install
Details
Feature Contributions
Changelog
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
cmd

```
Microsoft Windows [Version 10.0.22000.1219]
(c) Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\wolf1\Documents\Buecher\HTML5-Englisch\8117_Zusatzmaterial_Cod
C:\Users\wolf1\Documents\Buecher\HTML5-Englisch\8117_Zusatzmaterial_Cod
ebeispiele\html-beispiele.pronix.de\Beispiele\Kapitel013\13_3_6>
```

Figure 16.2 Finding and Installing Live Sass Compiler in Visual Studio

Next, I created a new folder in which I created an *index.html* file and a *styles* folder with the *style.scss* file in it. I’ve deliberately kept the *index.html* file simple.

```
...
<head>
    <title>Sass during execution</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="styles/style.css">
</head>

<body>
    <article class="my-article">
        <h1>Article 1</h1>
        <p>Lorem ipsum dolor … </p>
    </article>
    <article class="my-article">
        <h1>Article 2</h1>
        <p>Lorem ipsum dolor … </p>
    </article>
</body>
...
```

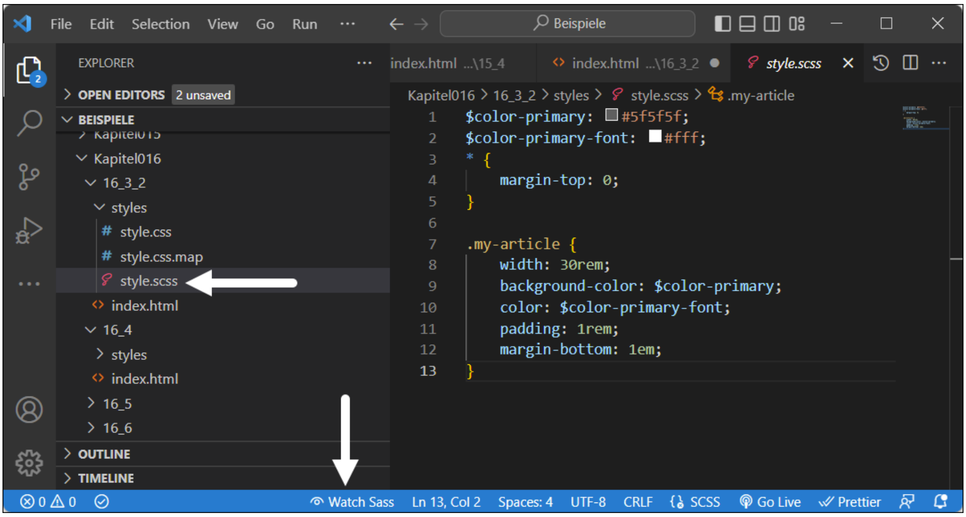
Listing 16.1 /examples/chapter016/16_3_2/index.html

In the *index.html* file, I've already included the CSS file `style.css`, which doesn't exist yet. We now want to style the `my-article` class using Sass. In the *style.scss* file, I wrote the following content:

```
$color1: #6d6d6d;
$color2: #fff;

.my-article {
    width: 30rem;
    background-color: $color1;
    color: $color2;
    padding: 2rem;
}
```

Listing 16.2 /examples/chapter016/16_3_2/style/style.scss

For the CSS preprocessor Sass to automatically make a CSS file named *style.css* out of it, all you need to do is enable the **Watch Sass** option at the bottom of the Visual Studio Code development environment. This enables a live translation from Sass or SCSS to CSS. Voilà, you've now created the CSS file from the SCSS file. That's all it took.



```
$color-primary: #5f5f5f;
$color-primary-font: #fff;
* {
    margin-top: 0;
}

.my-article {
    width: 30rem;
    background-color: $color-primary;
    color: $color-primary-font;
    padding: 1rem;
    margin-bottom: 1em;
}
```

File
Edit
Selection
View
Go
Run
Beispiele
EXPLORER
OPEN EDITORS
2 unsaved
BEISPIELE
Kapitel015
Kapitel016
16_3_2
styles
style.css
style.css.map
style.scss
index.html
16_4
styles
index.html
16_5
16_6
OUTLINE
TIMELINE
index.html ...\15_4
index.html ...\16_3_2
style.scss
Kapitel016 > 16_3_2 > styles > style.scss > .my-article
0 0
Watch Sass
Ln 13, Col 2
Spaces: 4
UTF-8
CRLF
SCSS
Go Live
Prettier

Figure 16.3 "Watch Sass" Allows You to Turn the SCSS File into a CSS File

No More Changes to the CSS File!

If you use Sass and have enabled the **Watch Sass** option, you shouldn't make any changes to the CSS file because you've now given the CSS preprocessor Sass control over it with the SCSS file. The changes to the CSS document would be overwritten when the Sass preprocessor is active and the SCSS file gets recompiled.

The great thing now is that as long as you have **Watch Sass** enabled, you don't have to worry about updating the CSS file. As soon as you make changes in the SCSS file, the CSS file will be automatically adjusted as well. You can recognize the active live translation when the **Watch Sass** label is replaced by the **Watching** label. The translation of the SCSS file takes place as soon as you save that file again.

16.3.3 Installing Sass for the Command Line

Of course, you can also install Sass from the command line. However, I'll only briefly describe this here. Before you can install Sass, you need Ruby on the computer. On macOS, Ruby is already integrated. On Windows, you must download and install the Ruby Installer from *https://rubyinstaller.org/*. On Linux, the following command on the command line is sufficient to install Ruby:

```
$ sudo apt-get install ruby
```

Once Ruby has been installed, you need to open the command line. On Windows, you open the **Start** menu and select **Start Command Prompt with Ruby**. On macOS and Linux, you want to launch the terminal. Then you must enter the following into the command line:

```
$ gem install sass
```

You may need to use `sudo` on macOS and Linux (`sudo gem install sass`). Then enter "`sass -v`" in the command line. If a version number gets displayed, you can be sure that Sass has been installed.

To compile a SCSS file, you need to change in the command line to the directory where you've saved the Sass file with the extension **.scss*. Then you can use the following command to compile the SCSS file to get a CSS file:

```
$ sass style.scss:style.css
```

In this example, you compile the SCSS file *style.scss* and get the CSS file *style.css*.

In addition to the compilation of Sass files, you can also set up a monitoring service for files or directories in the command line, so that once a change has been saved in the SCSS file, it automatically gets compiled, as described in the preceding section with

Visual Studio Code and the **Watch Sass** function. To monitor a single file, you need to use the following command in the command line:

```
$ sass --watch style.scss:style.css
```

This makes sure that when the SCSS file *style.scss* gets changed, it will automatically be compiled and the result will be saved in the CSS file, *style.css*.

You can monitor an entire folder via the following command:

```
$ sass --watch styles:styles
```

This will monitor all SCSS files in the *styles* directory and compile and save them as a CSS file in the same directory when a (saved) change occurs. For example, with an SCSS file named *layout.scss*, you would find a *layout.css* file in the *styles* directory after compilation. If you like it a bit neat, you can also create an *scss* folder and a *css* folder. In the *scss* folder, you store all SCSS files. Now, to save the CSS files in the *css* directory when compiling these SCSS files, you need to enter the following command:

```
$ sass --watch stylesheets/scss:stylesheets/css
```

Setting Output Styles for the CSS

You can also specify the output style for the CSS that Sass is supposed to create from the SCSS file. With the default value `nested`, everything is neatly nested. There are also `expanded` (indented), `compact` (everything in one line), and `compressed` (without spaces and line breaks) available. You can specify these output styles using `--style`. Here's how you can create a compressed style that's best suited for performance optimization:

```
$ sass --watch --style compressed stylesheets/scss:stylesheets/css
```

16.4 Using Variables with Sass

When you look at a larger CSS project, you'll notice that many values of CSS features, such as colors, spacing, dimension, and so on, are constantly repeated. If one wants to change these values afterward, an extensive search and replace of the values is often started. With variables in Sass, you can save yourself this work. They allow you to set a value for a CSS feature as a variable and then use it anywhere in the SCSS document. After compiling the SCSS file, these variables are replaced with the actual values in the CSS file.

You introduce a variable with the dollar sign ($). This is followed by the name of the variable without spaces in between. You can assign a value to the variable via a colon and end the line with a semicolon—basically the same way you assign a value to a CSS feature. The corresponding syntax looks as follows:

```
$variable-name: value;
```

Real-life examples could look like the following:

```
$color-primary: #5f5f5f;
$color-primary-font: #fff;
$font: 'Franklin Gothic','Arial Narrow',Arial,sans-serif;
$spacing-std: 1em;
```

You can then pass the variables defined in this way to a CSS feature using `$variable-name`. You're free to choose the name of the variable. Note, however, that you can't use any special characters. It's recommended to use a meaningful designation. A name like `$heaven-blue` isn't ideal because you might decide later to change the color from blue to yellow. Thus, a label like `$color-primary` is more appropriate. If you define a variable multiple times in the code, the last definition made always wins the deal when it comes to assigning a value to a CSS feature. You already know this from CSS as well.

Here's an example and what the CSS preprocessor makes of it.

SCSS File

```
$color-primary: #5f5f5f;
$color-primary-font: #fff;
$font: 'Franklin Gothic',
  'Arial Narrow',Arial, sans-serif;
$spacing-std: 1em;

* {
  margin-top: 0;
}

body {
  font-family: $font;
}

.my-article {
  width: 30rem;
  background-color: $color-primary;
  color: $color-primary-font;
  padding: $spacing-std;
  margin-bottom: $spacing-std;
}
```

SCSS file: */examples/chapter016/16_4/style/style.scss*

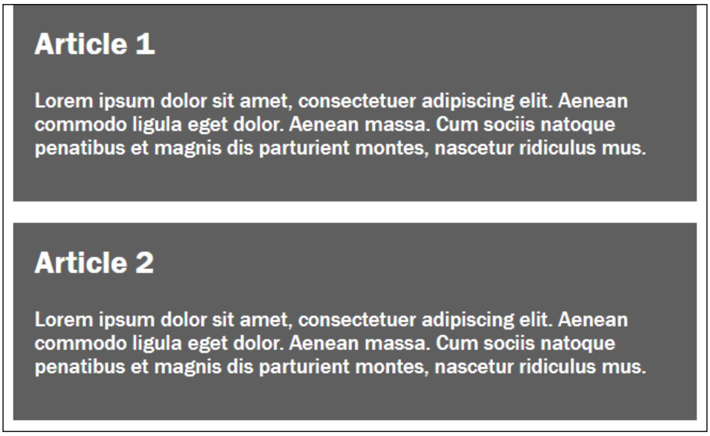
CSS File

```
* {
  margin-top: 0;
}

body {
  font-family: "Franklin Gothic",
  "Arial Narrow", Arial, sans-serif;
}

.my-article {
  width: 30rem;
  background-color: #5f5f5f;
  color: #fff;
  padding: 1em;
  margin-bottom: 1em;
}
```

CSS file: */examples/chapter016/16_4/style/style.css*

Table 16.3 The SCSS File and the CSS File after Compilation

The HTML file used is again */examples/chapter016/16_3_2/index.html* from Section 16.3.2.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Figure 16.4 The HTML File /examples/chapter016/16_4/index.html during Execution

16.5 Nesting with Sass

An enormous relief when writing SCSS documents is the use of a nesting of selectors (*selector nesting*), as they occur in the HTML document. However, it's important to keep the nesting within limits to avoid overloading the specificity of CSS, which can lead to a performance brake. I recommend a nesting of two to maximum three levels here. Here's a simple example of what such a nesting of selectors looks like and what the CSS preprocessor makes of it. Here, the selectors `p` and `h1` are written inside the `my-article` class.

SCSS File

```
...
.my-article {
  width: 30rem;
  background-color: $color-primary;
  color: $color-primary-font;
  padding: 0.1em;
  margin-bottom: $spacing-std;
  border-radius: 5px;
  h1 {
  padding-left: 0.5em;
  }
  p {
  background-color: $color-secondary;
```

CSS File

```
...
.my-article {
  width: 30rem;
  background-color: #5f5f5f;
  color: #fff;
  padding: 0.1em;
  margin-bottom: 1em;
  border-radius: 5px;
}

.my-article h1 {
  padding-left: 0.5em;
}
```

Table 16.4 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
    color: $color-secondary-font;
    padding: 1em;
  }
}
```

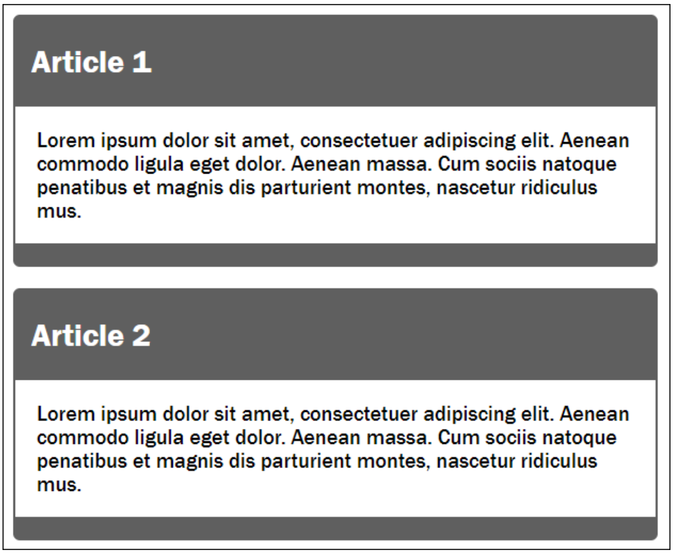
CSS File

```
.my-article p {
  background-color: #fff;
  color: #000;
  padding: 1em;
}
```

SCSS File	CSS File
SCSS file: */examples/chapter016/16_5/style/style.scss*	CSS file: */examples/chapter016/16_5/style/style.css*

Table 16.4 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

I think this kind of nesting significantly facilitates my work and also the overview in the SCSS document. The HTML example is the same as in the previous sections and now looks as shown in Figure 16.5.



Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Figure 16.5 The HTML File /examples/chapter016/16_5/index.html during Execution

Nesting is also useful for CSS properties (*property nesting*), which are grouped under a short name. For example, for the `padding` group, there are the individual properties `padding-top`, `padding-bottom`, `padding-left`, and `padding-right`. You know the same from other CSS properties such as `background-`, `margin-`, `border-`, `font-`, and more. Returning to the `padding` example, you can implement this CSS property group as a nested property in SCSS as follows.

SCSS File

```
.my-article {
...
  h1 {
  padding: {
  left: 0.5em;
  top: 0.1em;
  bottom: 0.1em;
  }
  }
...
}
```

/examples/chapter016/16_5/style/style2.scss

CSS File

```
...
.my-article h1 {
  padding-left: 0.5em;
  padding-top: 0.1em;
  padding-bottom: 0.1em;
}
...
```

/examples/chapter016/16_5/style/style2.css

Table 16.5 The SCSS File and the CSS File after the Preprocessor Run

16.6 Mixins (“@mixin”, “@include”)

Besides variables, *mixins* represent the most commonly used feature of Sass. Mixins are entire blocks of CSS features you can reuse as a whole at any time. Creating and using them is just as easy. To define a mixin, you want to follow these steps:

```
@mixin mixin-name {
   ...
}
```

As usual, you need to write your CSS features between curly brackets. Then, you can use `@include` to access this code block at any time:

```
@include mixin-name;
```

For this purpose, we’ll use our previous example from this chapter with mixins (see Table 16.6).

SCSS File

```
...
@mixin article-style {
  background-color: $color-primary;
  color: $color-primary-font;
  padding: 0.1em;
  margin-bottom: $spacing-std;
  border-radius: 5px;
}
```

CSS File

```
...
.my-article {
  width: 35rem;
  background-color: #5f5f5f;
  color: #fff;
  padding: 0.1em;
  margin-bottom: 1em;
  border-radius: 5px;
}
```

Table 16.6 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
@mixin article-content {
  background-color: $color-secondary;
  color: $color-secondary-font;
  padding: 2em;
}
...
.my-article {
  width: 35rem;
@include article-style;
  h1 {
  padding-left: 0.5em;
  }
  p {
@include article-content;
  }
}
```

SCSS file: */examples/chapter016/16_6/style/style.scss*

CSS File

```
.my-article h1 {
  padding-left: 0.5em;
}

.my-article p {
  background-color: #fff;
  color: #000;
  padding: 2em;
}
```

CSS file: */examples/chapter016/16_6/style/style.css*

Table 16.6 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

Those mixins are perfect for entire blocks of code that you use over and over again. As you can see in the example, the mixins also work with the variables of Sass. In the context of the variables, the desire quickly arises to make the mixins themselves a bit more flexible, for example, to use a mixin code block with different color combinations or other different values such as the text size. This isn't a problem with mixins because you can use them with arguments. Such arguments must be written inside parentheses with variables. An example will demonstrate the mixins with arguments (see Table 16.7).

In the example, you can see that you can pass values for the arguments as Sass variables or also with valid CSS values. The HTML file */examples/chapter016/16_6/index2.html* has been adapted in the corresponding position as follows:

```
...
    <article class="my-article">
        <h1>Article 1</h1>
        <p class="p1">Lorem ipsum ... </p>
    </article>
    <article class="my-article">
        <h1>Article 2</h1>
        <p class="p2">Lorem ipsum ... </p>
    </article>
...
```

Listing 16.3 /examples/chapter016/16_6/index2.html

SCSS File

```
...
@mixin article-content(
  $bg-color, $txt-color, $spacing) {
  background-color: $bg-color;
  color: $txt-color;
  padding: $spacing;
}
...
.my-article {
...
  .p1 {
  @include article-content(
  $color-secondary,
  $color-secondary-font,
  $spacing-std);
     }
  .p2 {
  @include article-content(
  yellow,
  $color-secondary-font,
  0.5em);
  }
}
```

SCSS file: */examples/chapter016/16_6/style/style2.scss*

CSS File

```
...
.my-article .p1 {
  background-color: #fff;
  color: #000;
  padding: 1em;
}

.my-article .p2 {
  background-color: yellow;
  color: #000;
  padding: 0.5em;
}
```

CSS file: */examples/chapter016/16_6/style/style2.css*

Table 16.7 The SCSS File and the CSS File after the Preprocessor Run

What’s missing now are mixins with default values for the arguments to still be able to call and use the mixin without special values. You can do this by writing the default value after the variable separated by a colon:

```
...
@mixin article-content($bg-color:$color-secondary,
                       $txt-color:$color-secondary-font,
                       $spacing:$spacing-std) {
    background-color: $bg-color;
    color: $txt-color;
    padding: $spacing;
}
...
.my-article {
...
    .p1 {
        @include article-content;
    }
    .p2 {
        @include article-content(yellow, $color-secondary-font, 0.5em);
    }
}
```

Listing 16.4 /examples/chapter016/16_6/style/style3.scss

In the example, I've used Sass variables as default values, which I can then adjust as needed. You can also just use a valid CSS value here. Now you can use the mixin with arguments and default values with and without values. If you don't set any values, the default values will be used. Here it's also possible that you call the mixin with one or two values. In that case, the default value will be used for the second and/or third value. For example, the following is also possible thanks to the default values:

```
@include article-content(green);
@include article-content(blue, yellow);
```

16.7 Extend ("@extend")

In addition to mixins, there's another way to avoid unnecessary repetitions, namely *extends*. It often happens with many selectors that they differ only by a few properties. The extends can be used in two different ways. First, I'll show you how to use `@extend` to split the CSS features of a selector and override or extend the necessary features in the new selector. In the example, we create the `.my-article` class selector as the base class for the other two class selectors, `.my-article-top` and `.my-article-std`, where we only change or override the color of the base class. Of course, you could also expand the two new classes. The extension is made here via `@extends selector name`. Any type of selector such as class selector, ID selector, element selector, and so on can be used as `selector name`.

SCSS File

```
...
.my-article {
  width: 35rem;
  background-color: $color-primary;
```

CSS File

```
.my-article,
.my-article-top,
.my-article-std {
  width: 35rem;
```

Table 16.8 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
  color: $color-primary-font;
  padding: 0.1em;
  margin-bottom: $spacing-std;
  border-radius: 5px;
  h1 {
  padding-left: 0.5em;
  }
  p {
  background-color: $color-secondary;
  color: $color-secondary-font;
  padding: 2em;
  }
}

.my-article-top {
@extend .my-article;
  background-color: darkslategray;
}

.my-article-std {
@extend .my-article;
  background-color: darkred;
}
```

SCSS file: */examples/chapter016/16_7/style/style.scss*

CSS File

```
  background-color: #5f5f5f;
  color: #fff;
  padding: 0.1em;
  margin-bottom: 1em;
  border-radius: 5px;
}

.my-article h1,
.my-article-top h1,
.my-article-std h1 {
  padding-left: 0.5em;
}

.my-article p,
.my-article-top p,
.my-article-std p {
  background-color: #fff;
  color: #000;
  padding: 2em;
}

.my-article-top {
  background-color: darkslategray;
}

.my-article-std {
  background-color: darkred;
}
```

CSS file: */examples/chapter016/16_7/style/style.css*

Table 16.8 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

Especially in an example like this one, you can see very clearly that with the help of `@extend`, the code in the SCSS document remains very clear, in contrast to the generated code in the CSS document. This is immensely useful in development. In the HTML document, the two classes `.my-article-top` and `.my-article-std` are used as follows:

```
...
    <article class="my-article-top">
        <h1>Article 1</h1>
        <p>Lorem ipsum ...</p>
    </article>
    <article class="my-article-std">
        <h1>Article 2</h1>
        <p>Lorem ipsum ...</p>
    </article>
...
```

Listing 16.5 /examples/chapter016/16_7/index.html

Because the base class `.my-article` doesn't appear at all in this example, we could have done without this selector because it unnecessarily inflates the CSS code. This takes us to the second option of using `@extend`.

Instead of defining a selector that won't be used at all and would only serve the extension with `@extend`, you can also use a placeholder. You introduce such a placeholder with `%`. Here again is the same example, but now `my-article` is degraded to a placeholder and no longer appears in the CSS code. This requires only one change in the code.

SCSS File

```
...
%my-article {
 width: 35rem;
  background-color: $color-primary;
  color: $color-primary-font;
  padding: 0.1em;
  margin-bottom: $spacing-std;
  border-radius: 5px;
  h1 {
  padding-left: 0.5em;
  }
  p {
  background-color: $color-secondary;
  color: $color-secondary-font;
  padding: 2em;
  }
}

.my-article-top {
@extend %my-article;
  background-color: darkslategray;
}
```

CSS File

```
.my-article-top,
.my-article-std {
  width: 35rem;
  background-color: #5f5f5f;
  color: #fff;
  padding: 0.1em;
  margin-bottom: 1em;
  border-radius: 5px;
}

.my-article-top h1,
.my-article-std h1 {
  padding-left: 0.5em;
}

.my-article-top p,
.my-article-std p {
  background-color: #fff;
  color: #000;
  padding: 2em;
}
```

Table 16.9 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
.my-article-std {
@extend %my-article;
  background-color: darkred;
}
```

SCSS file: */examples/chapter016/16_7/style/style.scss*

CSS File

```
.my-article-top {
  background-color: darkslategray;
}

.my-article-std {
  background-color: darkred;
}
```

CSS file: */examples/chapter016/16_7/style/style.css*

Table 16.9 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

In contrast to the preceding example, the class selector `.my-article` no longer appears.

“@mixin or” and “@extend”

Now that you know about two techniques to make code more modular and reusable with mixins and extends, the question is which of the two is the better choice. As for the use in the SCSS file, both are very suitable. However, the CSS code which gets created from it is also likely to be decisive. Especially with mixins, the code gets significantly inflated if there are a lot of repetitions. However, if you want to pass arguments, then you’ll prefer mixins over extends. My recommendation is to use `@extend` for SCSS code without arguments and to use `@mixin` for SCSS code with arguments. However, there are many different opinions on this. There are also developers who rely exclusively on mixins. This may also explain why mixins, along with variables, are often mentioned in an introduction to Sass, while the extends aren’t mentioned at all. Again, the following also applies here: There isn’t just one way of doing it. You now know two approaches.

16.8 Media Queries and “@content”

So that the mixins aren’t too much overshadowed after the extend section, I still want to bring the media queries into play here, for which mixins with arguments are again perfect: You can define the breakpoints as variables and adjust them at any time. It’s also useful that you can use inline media queries with Sass, which is an enormous help especially with extensive projects. The following example is intended to show how you can use media queries in Sass.

In this example, a mobile version (`30em`) and a desktop version (`60em`) were created. In the mobile version, the articles are displayed one below the other in the flexbox and side by side in the desktop version. In the created CSS code for the inline media query of Sass, you can also see that these media queries were created after compiling for each selector. This indeed means that there are a few more lines of code in the CSS file. But measured against the simplification and time saving of writing such media queries within the selector, the few bytes more than pay for themselves.

Besides the inline media queries, the use of variables with the mixins is also very useful, allowing you to customize the breakpoints quite comfortably. In addition, you can also apply arithmetic operators to the variables. I'll go into this separately.

You can also see in this example at `display: flexbox` that Sass also takes care of the browser prefixes.

SCSS File

```
...
$mq-mobile: 30em;
$mq-desktop: 60em;
@mixin breakpoint($mq-width) {
  @media screen and (
  min-width: $mq-width) {
  @content;
  }
}

.flex-container {
  display: flex;
  flex-flow: row wrap;
}

.my-article {
  font-family: $font;
  font-size: 1em;
  padding: 1em;
    ...
@include breakpoint($mq-mobile) {
 font-size: 1.125em;
  width: 90%;
  }
 @include breakpoint($mq-desktop) {
 font-size: 1.25em;
  width: 40%;
  }
}
```

CSS File

```
.flex-container {
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
  -webkit-box-orient: horizontal;
  -webkit-box-direction: normal;
  -ms-flex-flow: row wrap;
  flex-flow: row wrap;
}

.my-article {
  font-family: "Franklin Gothic",
  "Arial Narrow", Arial, sans-serif;
  font-size: 1em;
  padding: 1em;
}

@media screen and (min-width: 30em) {
  .my-article {
  font-size: 1.125em;
  width: 90%;
  }
}

@media screen and (min-width: 60em) {
  .my-article {
  font-size: 1.25em;
  width: 40%;
  }
}
```

Table 16.10 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
h1 {
  margin-top: 0;
@include breakpoint($mq-mobile) {
 font-size: 1.25em;
  }
@include breakpoint($mq-desktop) {
 font-size: 1.5em;
  }
}
```

CSS File

```
h1 {
  margin-top: 0;
}

@media screen and (min-width: 30em) {
  h1 {
  font-size: 1.25em;
  }
}

@media screen and (min-width: 60em) {
  h1 {
  font-size: 1.5em;
  }
}
```

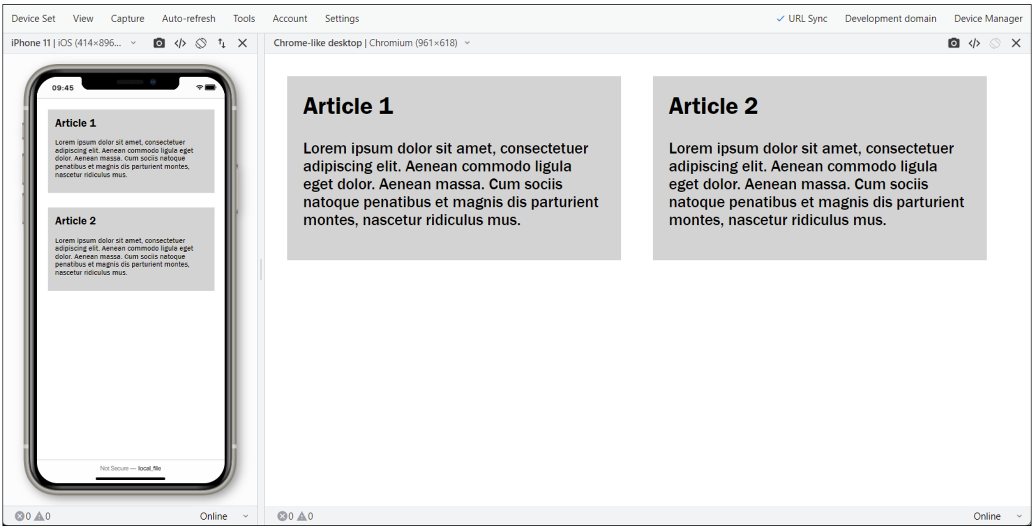
SCSS File	CSS File
SCSS file: */examples/chapter016/16_8/style/style.scss*	CSS file: */examples/chapter016/16_8/style/style.css*

Table 16.10 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

In the example with the mixin `breakpoint`, I still smuggled in `@content`, which works a bit like magic here:

```
...
@mixin breakpoint($mq-width) {
    @media screen and (
        min-width: $mq-width) {
        @content;
    }
}
...
```

As the name `@content` already describes, you can insert a content (into a mixin) with it. This instructs the CSS preprocessor to insert the contents of the subsequent SCSS code block at this point when compiling.


Device Set
View
Capture
Auto-refresh
Tools
Account
Settings
✓ URL Sync
Development domain
Device Manager
iPhone 11 | iOS (414×896...
Chrome-like desktop | Chromium (961×618)
09:45
Not Secure — local_file
Online
Online

Article 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Article 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Figure 16.6 Example in Execution: The Mobile Version and the Desktop Version in the Blisk Web Browser

16.9 Operators

As briefly mentioned in the previous section, you can also use the calculation operators +, -, *, /, and % (modulo) with Sass. It can be quite useful to have the values calculated during the design and layout. Here's an example that demonstrates a few ways you can use operators in Sass. I used this example in the context of media queries and adjusted a few properties with simple calculations.

SCSS File

```
...
$article-width: 80%;
$font-increase: 0.125em;
$base-size: 1;
...
.my-article {
  font-family: $font;
font-size: $base-size * 1em;
padding: $base-size * 1em;
  background-color: lightgray;
  margin: 1em;
  @include breakpoint($mq-mobile) {
font-size: $base-size +
  $font-increase;
```

CSS File

```
.my-article {
  font-family: "Franklin Gothic",
  "Arial Narrow", Arial, sans-serif;
  font-size: 1em;
  padding: 1em;
  background-color: lightgray;
  margin: 1em;
}

@media screen and (min-width: 30em) {
  .my-article {
  font-size: 1.125em;
  width: 80%;
  }
}
```

Table 16.11 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
  width: $article-width;
  }
  @include breakpoint($mq-desktop) {
font-size: $base-size +
  ($font-increase * 2);
width: $article-width / 2;
  }
}
```

CSS File

```
@media screen and (min-width: 60em) {
  .my-article {
  font-size: 1.25em;
  width: 40%;
  }
}
```

SCSS File	CSS File
SCSS file: */examples/chapter016/16_9/style/style.scss*	CSS file: */examples/chapter016/16_9/style/style.css*

Table 16.11 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

These examples don't really need much more description. The usual rules such as "multiplication and division before addition and subtraction" apply here as well. You may be a bit surprised by the notation `$base-size * 1em` because it only calculates 1 × 1. This is necessary here because the `$base-size` variable has no unit, and this way we set the unit `em`. This wasn't necessary for `$base-size + $font-increase` because `$font-increase` was assigned the unit `em`.

16.10 Adjusting Colors and Brightness

Colors can be specified in Sass as usual in CSS with `rgb()`, `rgba()`, `hsl()`, `hsla()`, `#fff`, `#ffffff`, and of course the CSS keywords for color such as `green`, `gray`, `red`, and so on. In addition, Sass contains functions to adjust the brightness and saturation of colors. I've listed some of these functions for you in Table 16.12.

Syntax	Example	Description
`lighten(color, [n]%)`	`background:lighten($color, 10%);`	Lightens the color by n%
`darken(color, [n]%)`	`background:darken($color, 10%);`	Darkens the color by n%
`desaturate(color, [n]%)`	`background:desaturate($color, 30%);`	Reduces the color saturation by n%
`saturate(color, [n]%)`	`background:saturate($color, 30%);`	Increases the color saturation by n%

Table 16.12 Some Useful Sass Functions for Manipulating Colors

Syntax	Example	Description
`adjust-hue(color, [n]%)`	`background:adjust-hue($color, -90%);`	Changes the hue of the color by n%
`invert(color)`	`background:invert($color);`	Inverts the color
`complement(color)`	`background:complement($color);`	Creates the complementary color to the specified color
`grayscale(color)`	`background:grayscale($color);`	Converts the specified color to grayscale

Table 16.12 Some Useful Sass Functions for Manipulating Colors (Cont.)

The functions are relatively easy to use. To give you an impression, I want to create an example with buttons. In it, I'll just define a color for the button and adjust the rest using the Sass color-manipulation functions. This way, you only need to change that one color, while you've created an entire theme system for a simple button.

The effort of the SCSS example seems at first more extensive than necessary when looking at the CSS code. However, now you've also created a great template here that you can use to generate many more buttons in different colors. You just need to create more classes in the style of `.my-btn`. Especially helpful in this example are the functions for color manipulation when the hover effect is applied and when the button is deactivated. When hovering, the color is merely darkened (`darken()`) and desaturated (`saturate()`). If the button gets disabled, it will be lightened (`lighten()`).

SCSS File

```
$btn-default: #3196cb;
$btn-color: white;

@mixin btn($btn-color:orange) {
  background: $btn-color;
  border-color: darken($btn-color, 10%);
}

@mixin btn-hover($btn-color:orange) {
$hover-color: saturate($btn-color, 10%);
  $hover-color: darken($hover-color, 10%);
  background: $hover-color;
border-color: darken($btn-color, 20%);
}
```

CSS File

```
.my-btn {
  margin-bottom: 1em;
  font-size: 14px;
  text-align: center;
  vertical-align: middle;
  cursor: pointer;
  padding: 0.5em 1em;
  border-radius: 4px;
  display: inline-block;
  border: 1px solid;
  color: white;
}

.my-btn {
  background: #3196cb;
  border-color: #2778a2;
}
```

Table 16.13 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
@mixin btn-disabled($btn-color:orange)
{
  background: lighten($btn-color, 20%);
border-color: lighten($btn-color,
10%);
}

%button-basic {
  margin-bottom: 1em;
  font-size: 14px;
  text-align: center;
  vertical-align: middle;
  cursor: pointer;
  padding: 0.5em 1em;
  border-radius: 4px;
  display: inline-block;
  border: 1px solid;
  color: $btn-color;
}

.my-btn {
  @extend %button-basic;
  @include btn($btn-default);
  &:hover {
  @include btn-hover($btn-default);
  }
  &.disabled,
  &.disabled:hover {
  cursor: not-allowed;
  opacity: .65;
  @include btn-disabled($btn-default);
  }
}
```

CSS File

```
.my-btn:hover {
  background: #1d7bac;
  border-color: #1d5979;
}

.my-btn.disabled, .my-btn.dis-
abled:hover {
  cursor: not-allowed;
  opacity: .65;
  background: #81c0e1;
  border-color: #58abd7;
}
```

SCSS File	CSS File
SCSS file: */examples/chapter016/16_10/style/style.scss*	CSS file: */examples/chapter016/16_10/style/style.css*

Table 16.13 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

In addition to the color manipulations, the ampersand character `&` was also placed in front of `:hover`, `.disabled`, and `.disabled.hover` in this example within the selector. The `&` is very useful when you use a nesting selector. For example, let's look at the following:

```
.my-btn {
  &.:hover {}
}
```

The CSS preprocessor turns this into the following:

```
.my-btn:hover {}
```

Without the ampersand character `&`, the CSS preprocessor would do the following:

```
.my-btn :hover {}
```

With this, the hover effect would not work. So, if you use the ampersand character inside a nested Sass selector, that selector will be appended to the parent selector instead of getting nested under it. This is especially popular in conjunction with the pseudo-class selectors such as `:hover` or `::after` because they need to be linked to a selector. With the `&` you can easily reference the parent selectors.

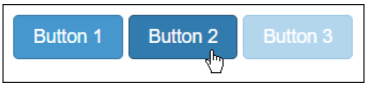


Figure 16.7 The HTML File /examples/chapter016/16_10/index.html during Execution

16.11 Sass Control Structures

In the previous example, you created a kind of Sass template for buttons. Using control structures such as loops, you can automate the generation of CSS code for buttons in different colors. In the example, I want to use the `@each` loop for this purpose. This loop makes it possible to process a list of elements. With regard to the buttons, a list of color values should be used in the form of CSS keywords. This way you can create different buttons in different color schemes quite comfortably and with very little effort. Of course, this can be applied to any other element as well. I've highlighted in bold the changes to the */examples/chapter016/16_10/style/style.scss* file from the previous section.

SCSS File

```
$btn-list: blue, darkred, darkgreen;
...
...
@each $btn-default in $btn-list {
  .my-btn-#{$btn-default} {
  @extend %button-basic;
  @include btn($btn-default);
```

CSS File

```
.my-btn-blue,
.my-btn-darkred,
.my-btn-darkgreen {
...
}
.my-btn-blue {
...
```

Table 16.14 The SCSS File and the CSS File after the Preprocessor Run

SCSS File

```
  &:hover {
  @include btn-hover($btn-default);
  }
  &.disabled,
  &.disabled:hover {
  cursor: not-allowed;
  opacity: .65;
  @include btn-disabled($btn-default);
  }
  }
}
```

CSS File

```
}
.my-btn-blue:hover {
...
}
.my-btn-blue.disabled,
.my-btn-blue.disabled:hover {
...
}
.my-btn-darkred {
...
}
.my-btn-darkred:hover {
...
}
.my-btn-darkred.disabled,
.my-btn-darkred.disabled:hover {
...
}
.my-btn-darkgreen {
...
}
.my-btn-darkgreen:hover {
...
}
.my-btn-darkgreen.disabled,
.my-btn-darkgreen.disabled:hover {
...
}
```

SCSS File	CSS File
SCSS file: */examples/chapter016/16_11/style/style.scss*	CSS file: */examples/chapter016/16_11/style/style.css*

Table 16.14 The SCSS File and the CSS File after the Preprocessor Run (Cont.)

The shortened CSS file demonstrates in a clear manner how further selectors for buttons with the different color schemes `.my-btn-blue`, `.my-btn-darkred`, and `.my-btn-darkgreen`, along with the hover and disable versions, were generated from the color specifications in the list. The comma-separated list `$btn-list` is passed through in the `@each` loop. The `$btn-default` variable is a placeholder and contains the respective color name of the `$btn-list`. On the first pass, `$btn-default` stands for `blue`, on the second pass for `darkred`, and on the third pass for `darkgreen`.

To ensure that the CSS color names are also attached to the selector, the `$btn-default` specification was interpolated with `#{}`. On the first pass, `.my-btn-#{$btn-default}` becomes `.my-btn-blue`, on the second pass `.my-btn-darkred`, and on the last pass `.my-btn-darkgreen`.

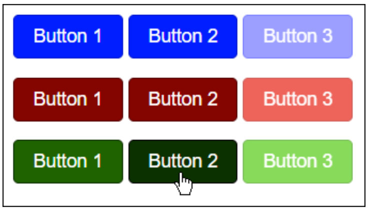


Figure 16.8 The HTML File /examples/chapter016/16_11/index.html with the Different Button Color Schemes during Execution

Admittedly, at this point, the example looks a bit more complex for starters, but it does nicely demonstrate that you can be extremely productive with Sass the deeper you get into it. Besides the `@each` loop, Sass provides other control structures as well.

In addition to the list-based loop `@each`, you'll find the classic loop `@for` in Sass, which allows you to specify a certain number of repetitions for the code. The `@for` loop comes in two flavors:

```
for $counter from 1 through 10 {
   // Code which gets executed 10 times
   .my-class-#{$counter} {
       ...
   }
}
```

The loop is executed 10 times here. The current loop pass is stored in `$counter`. Due to the interpolation with `#{}`, 10 selectors (`.my-class-1`, `.my-class-2`, etc.) are generated here. Besides the option `@for $ from [start]` **through** `[end]`, you can also use `@for $ from [start]` **to** `[end] end` here. With `through`, the final value is still executed, and, with `to`, the loop execution stops before it.

The `@while` loop is also available in Sass: such a loop is executed until the condition is false. However, you must be careful not to create an endless loop. Here's a pseudo-code for that:

```
$counter: 1;
$reply: 5;

@while $counter <= $reply {
  // Code that is executed until the condition is false, here 1<= 5
  // Increasing the $counter variable by 1
  $counter: $counter + 1;
}
```

Here, the loop is repeated as long as `$counter` is less than or equal to the value of `$reply`. Increasing the `$counter` variable at the end is very important because otherwise you would have an infinite loop.

Besides loops, there's also `@if`, which you can use to check a condition. The CSS preprocessor compiles a particular code only if the condition is true. To check conditions you can use different operators such as == (equal), != (unequal), > (greater than), < (less than), >= (greater than or equal), and <= (less than or equal).

In the following example, I've changed the `btn-hover` mixin. Optionally, you can now have buttons created without a hover effect if you pass a value other than 1 as a second parameter in addition to the color.

```
...
@mixin btn-hover($btn-color:orange, $hover-effect: 1) {
    @if $hover-effect==1 {
        $hover-color: saturate($btn-color, 10%);
        $hover-color: darken($hover-color, 10%);
        background: $hover-color;
        border-color: darken($btn-color, 20%);
    }
}
...
      @include btn-hover($btn-default, 0);
...
```

In this example, a CSS code block such as `.my-btn-blue:hover {...}` is only generated if `$hover-effect` is 1, which is the default value here.

Of course, there's also an alternative `@else` branch that gets executed when the `@if` condition doesn't apply. With regard to our example just shown, in the case of a disabled hover effect, the cursor should be changed to a stop symbol when you halt over the button with it. For example, you can use it to symbolize that this button can't be pressed.

```
...
@mixin btn-hover($btn-color:orange, $hover-effect: 1) {
    @if $hover-effect==1 {
        ...
    }
    @else {
        cursor: not-allowed;
    }
}
...
```

16.12 Functions "@function"

Sass also provides a way to create real functions by using `@function`. Unlike mixins, which output a section of code, functions return a return value. Different data types such as numeric values (`20`, `1.125`, `1.5em`), strings (`"text"`, `'text'`, `text`), colors (`#fff`, `#ffffff`, `rgba (255, 255, 0, 0.75)`), CSS value lists (`1em`, `1.5em`, `"Arial Narrow"`, `Arial`, `sans-serif;`), Boolean values (`false`, `true`), or even a null value (`null`) can be returned. A function is introduced with `@function` followed by the name of the function. The arguments of the function must be written between parentheses. Curly brackets must contain the code of the function, while `@return` will return a value. Here's a simple example that converts a pixel value to an `em` value:

```
$base-font-size: 16px;

@function px-to-em( $px-val ) {
    @return ( $px-val / $base-font-size ) * 1em;
}
```

You can use this function in the SCSS file as follows:

```
.my-article {
    width: px-to-em( 960px );
    font-size: px-to-em( 20px );
}
```

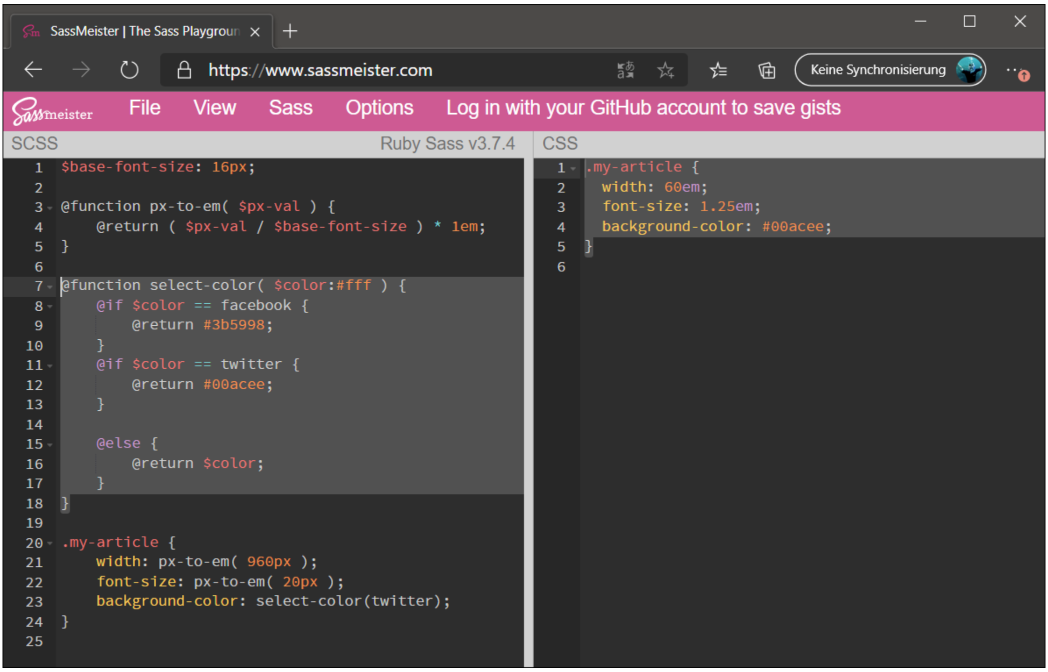
Then, the CSS preprocessor makes the following out of it:

```
.my-article {
  width: 60em;
  font-size: 1.25em;
}
```

Besides calculations, you can, of course, return other values from a function too. The following example uses a preferred color scheme such as Twitter or Facebook to return the color code. The list can be extended by any other color scheme.

```
@function select-color( $color:#fff ) {
    @if $color == facebook {
        @return #3b5998;
    }
    @if $color == twitter {
        @return #00acee;
    }

    @else {
        @return $color;
    }
}
```

I recommend you try out such mini samples directly in *www.sassmeister.com*. It’s the ideal playground to learn Sass without immediately using the individual techniques in a project.


SassMeister | The Sass Playgroun
https://www.sassmeister.com
Keine Synchronisierung
Sassmeister
File
View
Sass
Options
Log in with your GitHub account to save gists
SCSS
Ruby Sass v3.7.4
CSS

```
$base-font-size: 16px;

@function px-to-em( $px-val ) {
    @return ( $px-val / $base-font-size ) * 1em;
}

@function select-color( $color:#fff ) {
    @if $color == facebook {
        @return #3b5998;
    }
    @if $color == twitter {
        @return #00acee;
    }

    @else {
        @return $color;
    }
}

.my-article {
    width: px-to-em( 960px );
    font-size: px-to-em( 20px );
    background-color: select-color(twitter);
}
```

```
.my-article {
  width: 60em;
  font-size: 1.25em;
  background-color: #00acee;
}
```

Figure 16.9 Sassmeister Is Perfect for Learning Sass without Having to Use It in the Project Right Away

16.13 “@import”

When the projects become more extensive, it isn’t advisable to write everything in an SCSS file. As with CSS, in Sass, you can split SCSS files into meaningful sections such as reset, setup, layout, navigation, and so on; import them via `@import`; and compile all the spun-off sections into SCSS files to create a CSS file. As a web designer, you can still keep track of everything thanks to the individual SCSS files. For example, I personally use a *style.scss* file into which I import all other SCSS files. Here’s an example of using `@import`:

```
@import "reset";
@import "layout";
@import "basic";
```

Listing 16.6 /examples/chapter016/16_13/style/style.scss

When importing with `@import`, the file name between "" is sufficient. In this example, the SCSS files *_reset.scss*, *_layout.scss* and *_basic.scss* are compiled, and the entire content is written to the CSS file, *style.css*. The underscore at the start of the individual SCSS file names is important so that they don't get compiled directly. If you didn't use an underscore, you would have generated the *reset.css*, *layout.css*, and *basic.css* files in addition to the *style.css* CSS file. However, this is superfluous here and can be bypassed quite elegantly with the underscore before the file name. Of course, you can also specify the full file name when importing (even if it isn't necessary):

```
// File: style.scss -> style.css
@import "_reset.scss";
@import "_layout.scss";
@import "_basic.scss";
```

The order in which you write the SCSS documents when importing them is also of enormous importance, especially if you use variables that you also use in other documents. For a variable to be assigned correctly, it must also be known. For this purpose, it can be useful, for example, to write the variables in an extra SCSS file and import that file right at the beginning:

```
@import "reset";
@import "setup"; // All variables go here
@import "layout";
@import "basic";
```

16.14 Comments

Finally, I want to share a few words about comments in Sass. Especially if the code becomes more extensive, you'll probably want to use comments. Here you can proceed like you would in CSS and add a comment like the following in the SCSS file:

```
/* I am a comment */
```

The CSS preprocessor will also include this comment in the CSS file. In addition, Sass still provides the option to write a comment as follows:

```
// I am a comment
```

However, this comment can only span one line. Furthermore, this comment won't be added to the CSS file by the CSS preprocessor. I find this useful as a method to separate comments that are significant for Sass from general comments for CSS.

16.15 Summary

In this chapter, you learned the basics of Sass and SCSS. Of course, that's not all this CSS preprocessor has to offer. I therefore recommend that you first familiarize yourself somewhat with what you've learned and gain a little practical experience. Often, you'll have your "aha" experience when you see what kind of CSS code the CSS preprocessor generates. For beginners, I'd recommend to limit yourself to basic things such as variables, mixins, and extends for the time being.

If you want to delve deeper into the subject, then *http://thesassway.com* is a good source. In addition, as you get more involved with Sass, you'll quickly realize that there are many extensions available for it. One of the most popular representatives here is Compass (*http://compass-style.org*).

Chapter 17

A Brief Introduction to JavaScript

JavaScript, Ajax, and jQuery are often mentioned in the same breath in the context of web development. For beginners, it's often frustrating to be confronted with many different terms. In this chapter, you'll first get to know some basics about JavaScript as a programming language.

When it comes to web development, JavaScript has become indispensable. While you can use HTML to create the content of your website and CSS to design the layout and formatting, you're still missing a way to dynamically influence the behavior of a website within the web browser. JavaScript enables you to perform Document Object Model (DOM) manipulations such as changing HTML elements, HTML attributes, and HTML styles, as well as checking entered data in HTML forms for correctness. Let's not forget the now numerous JavaScript application programming interfaces (APIs; also called web APIs) in HTML. Even for the use of many frameworks, such as React, Angular, or Vue.js, you can't get around sound JavaScript knowledge.

This chapter is intended to give you a basic and simple introduction to the world of JavaScript. To avoid raising false hopes here, I should mention that this chapter will only introduce you to JavaScript as a scripting language. JavaScript is a programming language that can't be described quickly in its entirety within one chapter. The introduction to JavaScript in this book only goes so far as to let you use JavaScript for client-side applications of the DOM, the interfaces between HTML, and dynamic JavaScript—more specifically—you'll learn how to write programs that run in the web browser.

Not only is JavaScript now suitable for client-side applications, as described in this book, but the language has become very versatile. For example, JavaScript is also used today for server-side applications, desktop applications, mobile applications, and even embedded applications. Even games and 3D applications can now be developed with JavaScript. However, this is only mentioned here in passing to show you that, with JavaScript, you learn a fairly ubiquitous language that can be used not only in the web browser.

For the professional handling of JavaScript I recommend you read *JavaScript* (SAP PRESS, 2022) by Philip Ackermann, who was also an expert reviewer for this book.

17.1 JavaScript in Web Development

The JavaScript language has been around since 1995 and has been constantly evolving ever since. In the beginning, language was seen more as a toy that could be used for all kinds of mischief. JavaScript only really got going as years passed. With Ajax, the language experienced a real boom, and there were first meaningful applications that would have been impossible without JavaScript on the client side.

JavaScript is a genuine and ubiquitous programming language, and if you've never programmed in another language such as PHP, Java, or C++ before, this is probably your first real programming language that you'll learn here. If you already have experience in another programming language, this chapter will be easy for you.

JavaScript Is an Interpreted Programming Language

JavaScript is an interpreted programming language where the source code is executed by an interpreter on the computer. The interpreter, in turn, converts the source code into machine code, statement by statement. Usually, you don't have to worry about anything. You can write the source code with any text editor, and the interpreter is provided and executed in the web browser.

The counterpart to an interpreted programming language is the compiled programming language. Here, the source code is translated into machine code by a compiler. You can then run the programs created in this way on the operating system for which you compiled it, without any further tools (e.g., an interpreter). Examples of a compiled language include C, C++, or Swift.

With interpreted programming languages such as JavaScript, the program can thus be executed directly and doesn't need to be recompiled each time. However, this also means that a syntax error in interpreted languages is often detected only at program runtime. The performance of compiled programming languages is usually somewhat better because no more source code has to be converted at runtime. However, with interpreted programming languages, a just-in-time compiler (JIT) is used, which converts frequently executed source code into machine code that is then executed faster.

JavaScript allows you to access the HTML document displayed in the web browser and respond to user input, for example. In response to user input (e.g., a button has been pressed), you can make content or presentation-specific changes to the HTML document. The change applies only to the HTML document in the memory while the HTML file on the web server remains untouched. This dynamic read access to the HTML document is provided by the DOM.

Here are a few lines about where JavaScript fits in between HTML and CSS in modern web design. As you already know, HTML is used for structured content. CSS is responsible for the presentation. HTML is thus the foundation for web development on top of which CSS is built. And in the same way as CSS is built on top of HTML, you can view

JavaScript. JavaScript is also built on top of HTML (in web development) and is responsible for the behavior of the website (more precisely, the interaction). JavaScript extends HTML in the sense of bringing dynamics to websites. To avoid any misunderstanding, JavaScript doesn't extend the HTML language. Where CSS improves the presentation of the website and thus the overview, JavaScript aims to improve the usability of the document with a specific behavior on an interaction.

LiveScript, JavaScript, JScript ECMAScript, ECMA

JavaScript was developed by Netscape under the name LiveScript for Netscape Navigator 2 and only later was renamed to JavaScript. In what's referred to as the web browser war between Netscape and Microsoft at the time, Microsoft also designed a similar scripting language to JavaScript: JScript. With two scripting languages now in circulation—JavaScript and JScript—it was time for JavaScript to be standardized. This standard was quickly found and called ECMAScript. Today, the ECMA (European Computer Manufacturers Association) defines the core of the JavaScript specification. The versions were numbered from 1 (ECMAScript 1) to 6 (ECMAScript 6) until 2015. Since 2015, the year gets appended to the name (ECMAScript 2015). When this book went into print, ECMAScript 2022 was the latest version. We can probably expect ECMAScript 2023 in the summer of 2023.

In Figure 17.1, you can see a basic model on the basis of which modern websites are created. Apart from HTML for structuring content, the other techniques aren't mandatory for a website to work. So, you can create websites that contain only HTML and CSS, just as you can create websites that use only HTML and JavaScript. In common practice, you'll mainly use a combination of all three web techniques with HTML as the base and CSS and JavaScript as add-ons, but you can also use HTML on its own. Web developers also talk about the three layers: Content layer (HTML), presentation layer (CSS), and behavior layer (JavaScript).

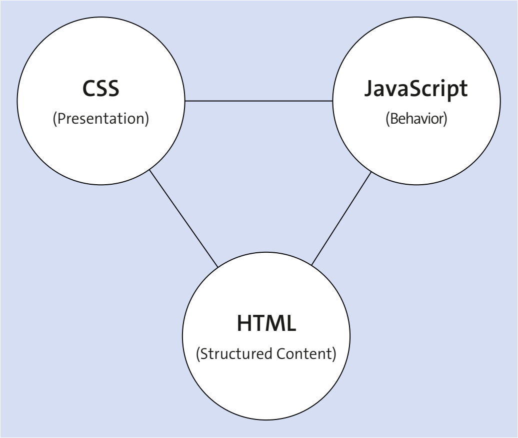


Figure 17.1 Building Blocks of a Modern Website

Like HTML and CSS, JavaScript is readable in plain text and can be written in an ordinary text editor. The JavaScript runtime is already built into every web browser, so you don't need any additional tools.

JavaScript Engine

While you'll be using the web browser as the runtime environment for your JavaScripts in this book anyway, I still want to make you a little more aware of the topic so that you don't perceive JavaScript as just an ordinary part of a web browser. As you can guess, different browser vendors use their own runtime environment for JavaScript. For example, Google uses a V8 engine written in C++, which is used in Google Chrome. The latest Microsoft Edge now also uses Google's V8 engine. Firefox, on the other hand, uses a JavaScript runtime written in C called SpiderMonkey. Based on the SpiderMonkey runtime environment, additional modules have been added over time, primarily to improve performance. Apple also uses its own runtime environment, JavaScriptCore (also called Nitro), which is used in the Safari web browser, for example.

17.2 Writing and Executing JavaScript Programs

In this section, I'll show you how to write and run JavaScript in web development. The JavaScript code itself isn't yet the focus at this point. As a tool for development, I recommend that you use an editor that can highlight JavaScript syntax and also detect syntax errors in the source code. This is enormously useful when you develop JavaScript code. I also use Microsoft's Visual Studio Code for that, which is available for all platforms. But there are other exciting alternatives, such as Sublime Text (*http://sublimetext.com*), Notepad++ (*https://notepad-plus-plus.org*), or Nova 2 (*https://nova.app*).

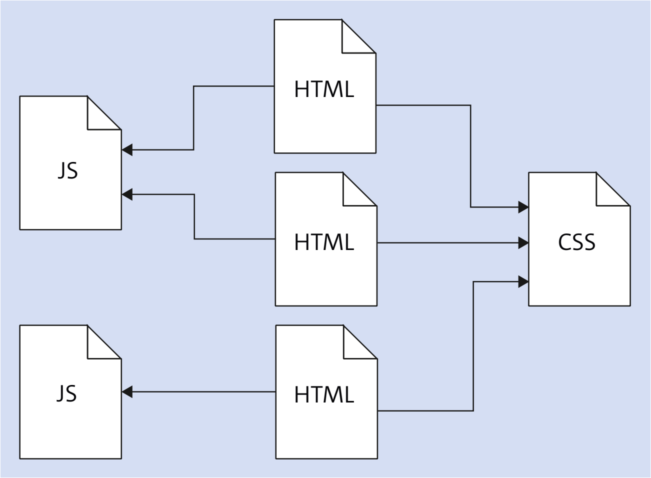


Figure 17.2 For the Reusability of Longer JavaScript Code, It's Recommended to Store It in a Separate JavaScript File in Addition to a CSS and HTML File

When writing JavaScript code, it's also advisable to separate the individual layers, that is, save HTML, CSS, and JavaScript in a separate file. This way, you can reuse the same JavaScript source code in different HTML files.

17.2.1 Integrating a JavaScript File in an HTML File

The first example in JavaScript will simply output a tip box with the text "Hello JavaScript". The goal of the example is initially just to show you how to use JavaScript in HTML files. I usually prepare a folder named *js* or *scripts* for this, where I store the JavaScript source code.

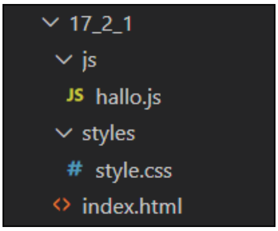


Figure 17.3 A Clean Folder Structure Helps You Keep Track of More Extensive Projects

Here's a simple listing, which I'll call *hello.js*, and store in the *js* directory. The recommended file extension for JavaScripts is *.js*. Although you can use other endings here, editors and browsers will know right away what the content is about.

```
function showHello() {
    alert('Hello JavaScript!');
}

// Call showHello() function
showHello();
```

Listing 17.1 /examples/chapter017/17_2_1/hello.js

The example defines a function with the identifier `showHello` that calls a JavaScript built-in function `alert` with the text "Hello JavaScript!". The `showHello` function alone wouldn't have any effect and must be called somewhere in the JavaScript, which is done here at the end of the example with `showHello();`. The `alert` function prints the text passed between the parentheses in a tip dialog.

I already briefly mentioned in Chapter 3, Section 3.7 that you can integrate a JavaScript in an HTML document by means of the `script` element. You can use this `script` element in the head (`head` element) and in the displayable body (`body` element) of the HTML document. Most of the time, it's better to include the JavaScript right before the closing `body` element because then the entire DOM is loaded before the JavaScript starts running. You can either write the JavaScript code directly between the opening

`<script>` and the closing `</script>`, or—the recommended variant—the `script` element remains empty and you use the `src` attribute to reference an external file with a JavaScript code:

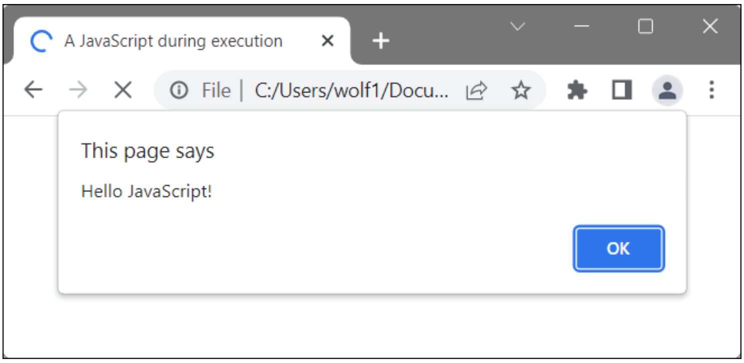
```
<script src="js/hello.js"></script>
```

To do this, you now want to create an HTML file named *index.html* and include the JavaScript *hello.js* before the end of the `body` tag as follows:

```
<!doctype html>
<html>
<head>
    <title>A JavaScript during execution</title>.
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="styles/style.css">
</head>
<body>
    <main>
        <article>
            <h1>Hello JavaScript!</h1>
            <p>Lorem ipsum ... </p>
        </article>
    </main>
    <script src="js/hello.js"></script>
</body>
</html>
```

Listing 17.2 /examples/chapter017/17_2_1/index.html

Once you've loaded *index.html* into the web browser, the tip dialog appears, which you can confirm via the **OK** button. The dialog looks different from web browser to web browser.



This page says

Hello JavaScript!

Figure 17.4 The JavaScript "hello.js" during Execution (Here, Microsoft Edge)

Integrating Multiple JavaScript Files in the HTML Document

You can integrate additional JavaScript files in an HTML file at any time. To do this, you just need to use a separate `script` element for each file.

17.2.2 Writing JavaScript within HTML

As mentioned earlier, you can also write the JavaScript code directly between the opening `<script>` and closing `</script>` in an HTML document as follows:

```
<script>
// JavaScript code;
...
</script>
```

However, you should use this method only in rare exceptional cases because this approach mixes JavaScript code and HTML code in one file. This may not really matter for a short JavaScript, but JavaScript source code can also become quite extensive. What is more, the JavaScript code can then not really be reused.

But anyway, here's the HTML file *index.html* again with directly written JavaScript code between the opening `<script>` tag and the closing `</script>` tag. The example does the same as the one before and outputs a tip dialog that reads “Hello JavaScript!”.

```
<!doctype html>
<html>
<head>
    <title>A JavaScript during execution</title>.
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="styles/style.css">
</head>
<body>
    <main>
        <article>
            <h1>Hello JavaScript!</h1>
            <p>Lorem ipsum ... </p>
        </article>
    </main>
    <script>
       function showHello() {
         alert('Hello JavaScript!');
       }
```

```
        // Call showHello() function
        showHello();
    </script>
</body>
</html>
```

Listing 17.3 /examples/chapter017/17_2/index.html

Integrating a JavaScript and Source Code between <script> and </script>

You can't simultaneously include a JavaScript with `src` and write JavaScript code between `<script>` and `</script>`. If you do that, the source code between `<script>` and `</script>` will be ignored, and the integrated JavaScript will be executed.

17.2.3 Position of JavaScript and Its Execution in the HTML Document

I need to discuss the position of the JavaScript in the HTML document because, in the past, this file was often included in the `head` section. You might expect it there too because you also include linked CSS files there, for example. By the way, there's nothing fundamentally wrong with including the JavaScript in the header of the HTML document. To understand why you should still prefer to include a JavaScript at the end of the HTML document, you need to know how `script` elements are executed in an HTML document.

When a web browser receives the HTML document from the web server, it usually starts processing the HTML code with a parser to create a certain structure, the DOM, from it. When this parser encounters a `script` element, it stops processing, and the JavaScript code inside the `script` element gets executed. After executing the JavaScript code, the parser continues to process the rest of the HTML document. The following is the result of this way of processing JavaScript code in HTML documents:

- **Slow page load due to external scripts**
  The fact that embedded scripts are executed while the HTML document is being read usually also slows down the page load when larger JavaScript files have to be downloaded from the web server. You can avoid the problem by writing the `script` elements at the end of the document before the closing `<body>` tag, if possible. Although the JavaScript must still be downloaded before execution, the fact that the HTML page is already displayed in the web browser means that it takes less time to build the web page.
- **Access only to loaded elements**
  Because a JavaScript interrupts the execution of the parser and has access to the DOM tree, JavaScript can only access this DOM tree as far as the parser has already processed. You can't use the JavaScript to access elements in the DOM tree that have

yet to be processed by the parser. Again, it's important at which position you place the `script` element. You can use JavaScript to access only those elements that precede the written `script` element in the document. Everything else further down in the HTML document doesn't yet exist for the JavaScript. In addition, the `DOMContentLoad` event should be mentioned here, which is triggered when the DOM has been completely loaded.

- **Access to previously included resources**
  As you already know, you can use multiple `script` elements in the HTML document. You can mix external JavaScript files and code written in the document as you like. Because the `script` elements are executed in the order in which you wrote them down, you could write different scripts that build on each other. Thus, scripts included later via the `script` element could use previous resources (e.g., variables, objects, or functions) from scripts previously included with the `script` element.

17.2.4 Attributes for Manipulating the Load Behavior of JavaScript ("async", "defer")

At this point, the standalone attributes `async` and `defer` for the `script` element should be mentioned, which enable youto manipulate the load behavior. Both attributes make sense only if you include a JavaScript with `src`.

- **`async`**
  This attribute ensures that the download of the JavaScript file is asynchronous. This means that the processing of the HTML code doesn't pause. However, it also makes sure that the JavaScript code is executed directly as soon as the JavaScript file has been downloaded. For this reason, this attribute is only suitable for scripts that work independently of the HTML document.
- **`defer`**
  This attribute also ensures that the processing of the HTML code doesn't pause. However, with `defer`, unlike `async`, the JavaScript code won't get executed until the HTML code has been completely processed. Only then will the JavaScript code be executed. The attribute was often avoided because older web browsers weren't able to handle it. However, `defer` is no longer a problem for current web browsers.

17.2.5 The <noscript> Element for No JavaScript

If the visitor has JavaScript disabled or if the web browser doesn't support JavaScript, you can place a special note between `<noscript>` and `</noscript>`. Here's a simple example of this:

```
...
    <noscript>
        JavaScript is not available or is disabled. <br />
        For optimal use of this website it is recommended
        to use a browser with JavaScript or
        to enable JavaScript in your browser.
    </noscript>
...
```

Listing 17.4 /examples/chapter017/17_2_5/index.html

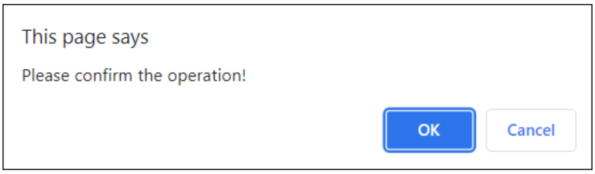
You shouldn't overload your website unnecessarily with the `noscript` element, but use it only to inform visitors about the options that are available to them when JavaScript is enabled. If you need to use a lot of `noscript` elements, you should rethink the structure of your website. The most important information on your website should be accessible without JavaScript. Using JavaScript, you simply add additional functionality to the web page that may improve its operation. Many web developers make little use of the `noscript` element unless it's a web page that was written as a pure JavaScript application and requires a JavaScript-enabled web browser to function.

17.3 JavaScript Output

Previously, you used a standard `alert()` dialog for JavaScript output. In addition to that dialog, there are other ways to generate output. In this section, you'll learn what other options you have and where you can use them.

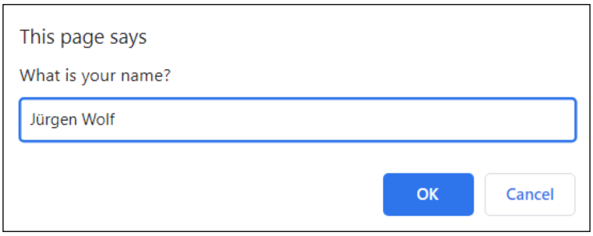
17.3.1 Standard Dialogs (and Input Dialog)

In addition to the standard dialog `alert()`, there are two more dialogs: `confirm()` and `prompt()`. The `confirm()` dialog is the classic OK-cancel dialog with two buttons, while `prompt()` serves as a dialog for entering text. However, these standard JavaScript dialogs are rarely used in practice because their layout depends on the underlying web browser, and the options to apply them are rather limited. Furthermore, the standard dialogs have the disadvantage that the web browser can ignore them if they are displayed repeatedly. You can find an example of the two dialogs `confirm()` and `prompt()` in */examples/chapter017/17_3_1/index.html*.


This page says

Please confirm the operation!

Figure 17.5 The Standard Dialog confirm() (in Google Chrome)



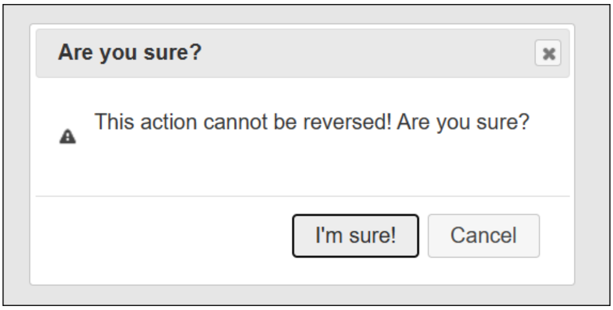
This page says

What is your name?

Jürgen Wolf

Figure 17.6 The Standard Dialog prompt() (in Google Chrome)

In practice, it's better to use ready-made JavaScript libraries or frameworks that provide dialogs which match the design.



Are you sure?

This action cannot be reversed! Are you sure?

Figure 17.7 The Dialog Was Created Using the jQuery UI Library and Looks the Same in any Web Browser

17.3.2 Outputting to the Console

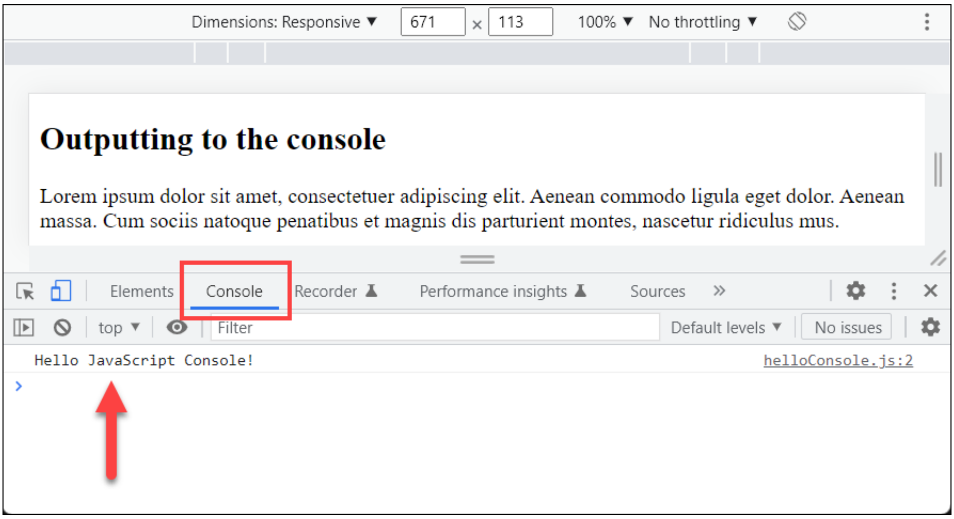
The easiest way to write something in the web console is to use the `console` object provided by the runtime environments of the web browsers you use. Although the `console` object isn't included in the standard ECMAScript, it's supported by any JavaScript runtime environment.

`log()`, which is a function (or method) of the `console` object, allows you to generate a simple console output. However, such a log output in the console isn't intended for the users, but should help you as a JavaScript developer to understand the course of individual program sections or to track down errors. Here's a simple example that demonstrates the `log()` method in use.

```
function showConsole() {
    console.log('Hello JavaScript Console!');
}
showConsole();
```

Listing 17.5 /examples/chapter017/17_3_2/js/helloConsole.js

You include the JavaScript as usual in the HTML document by using the `script` element. When you call the web page, basically nothing happens because the JavaScript uses the console as output. For this reason, you need to open the console of the corresponding web browser. You can find it among the developer tools or development tools of the web browser, which you can often call via `Ctrl` + `Shift` + `I`. There, you'll usually also find the **Console** tab with the output of the JavaScript.



Dimensions: Responsive ▼
671 × 113
100% ▼
No throttling ▼
Elements
Console
Recorder
Performance insights
Sources
top ▼
Filter
Default levels ▼
No issues

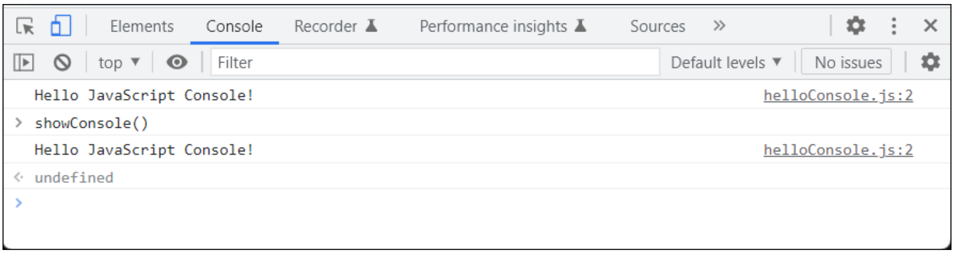
Outputting to the console

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

```
Hello JavaScript Console!                                        helloConsole.js:2
>
```

Figure 17.8 The Output of the JavaScript to the Console of the Web Browser

You can do more than just check the output of JavaScripts in the web browser console. You can also make entries. For example, for testing, you can also type the `showConsole()` function and manually execute this function from *helloConsole.js*. Of course, this also requires the web page to be running and the JavaScript to be loaded. From this point of view, the console window is an important tool for web developers, so I recommended studying it a bit more.



Elements
Console
Recorder
Performance insights
Sources
top ▼
Filter
Default levels ▼
No issues

```
  Hello JavaScript Console!                                      helloConsole.js:2
> showConsole()
  Hello JavaScript Console!                                      helloConsole.js:2
< undefined
>
```

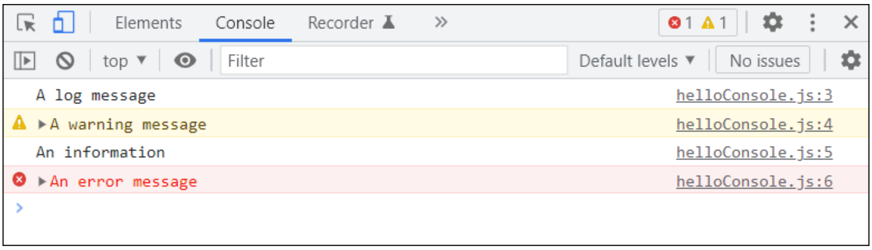
Figure 17.9 The Console Is Often Used during Development for Quick Outputs

Logging Outputs Only during Development!

You should only use the output via the `console` object during development. Even though visitors usually don't see these outputs anyway, you should stop using outputs to the console when the website is finished. For this reason, many web developers use special logging libraries, which can be used to switch logging information on or off at any time with the appropriate configuration.

Many runtime environments provide other categories of output besides `console.log()`. However, not all runtime environments provide all functions. Nevertheless, in addition to `console.log()`, `console.info()`, `console.warn()`, and `console.error()` are often available. All functions can be used like `console.log()`, except that the output in the web console often changes style, which can be useful if you want to filter out (error) messages depending on the category. You can also type the examples directly into the console for testing purposes:

```
console.log('A log message');
console.warn('A warning message');
console.info('An information');
console.error('An error message');
```



```
A log message                  helloConsole.js:3
▸A warning message             helloConsole.js:4
An information                 helloConsole.js:5
▸An error message              helloConsole.js:6
```

Figure 17.10 The Individual Outputs in the Console Usually Differ Visually

17.3.3 Outputting to the Website

The output with standard dialogs isn't really elegant, and the output via the `console` object is only for web developers. In practice, therefore, you'll often generate output for users of the website. In that case, there's no way around the HTML elements or attributes. For example, you can do anything from a classic input/output directly into a text field to a DOM manipulation where you dynamically modify a web page or parts of it at runtime to produce output. How you can do this with JavaScript is covered separately in Chapter 18.

The JavaScript in Listing 17.6, included in the HTML document in Listing 17.7, shows such an example. As soon as you click the `button` via the `button` element, a `click` event gets triggered, whereupon the JavaScript with the `changeText()` function gets executed via `onClick="changeText()"`. The function changes the content of the first `p` element it finds and replaces the text in between with `"The button was pressed! (1x)"`, where the value of `counter` is increased by 1 after each new confirmation:

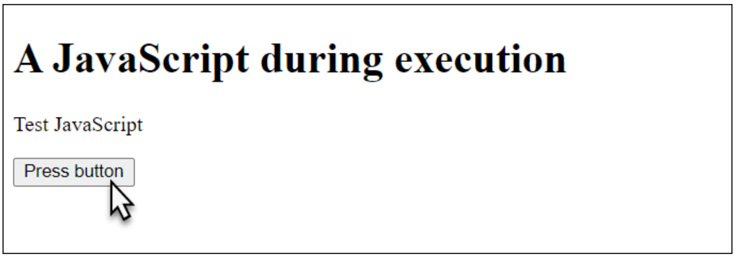
```
let counter = 1;
function changeText() {
    document.querySelector('p').innerHTML =
        "The button was pressed! (" + counter + "x)";
    counter++;
}
```

Listing 17.6 /examples/chapter017/17_3_3/js/pushButton.js

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>JavaScript test</title>
</head>

<body>
    <h1>A JavaScript during execution</h1>
    <p>Test JavaScript</p>
    <button type="button" onclick="changeText()">
      Press button
    </button>
    <script src="js/pushButton.js"></script>
</body>
</html>
```

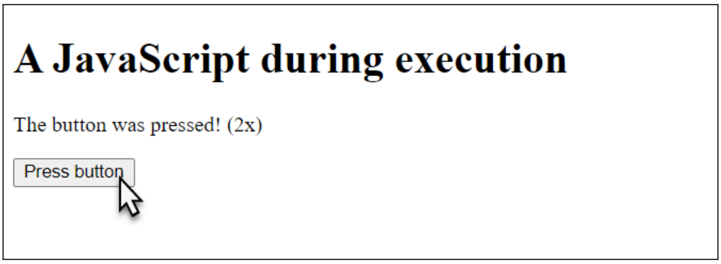
Listing 17.7 /examples/chapter017/17_3_3/index.html

At this point, you don't have to worry about the JavaScript code and how to use it in the HTML document. This example simply demonstrates a classic web development process of using JavaScript to access HTML elements to customize output for users.


A JavaScript during execution

Test JavaScript

Figure 17.11 The HTML Document with a Button



A JavaScript during execution

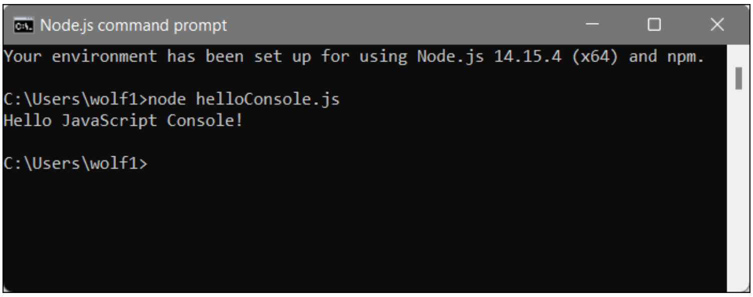
The button was pressed! (2x)

Figure 17.12 When the Button Is Pressed, a JavaScript Gets Executed That Manipulates the Content of the First <p> Element It Finds

17.3.4 Running JavaScript without a Web Browser

In the following sections and the next chapter, you'll mainly use pure JavaScript code to learn the language. The output is predominantly done by using `console.log()`. Instead of using an HTML document here where you include the JavaScript with the `script` element for execution, only to then view the output in the console, there's a more convenient way to run JavaScript without a web browser. For this purpose, you need to install Node.js (from *https://nodejs.org*) on your machine. Once you've installed Node.js, you can execute a JavaScript in the command line using the `node` command as follows:

```
$ node script.js
```



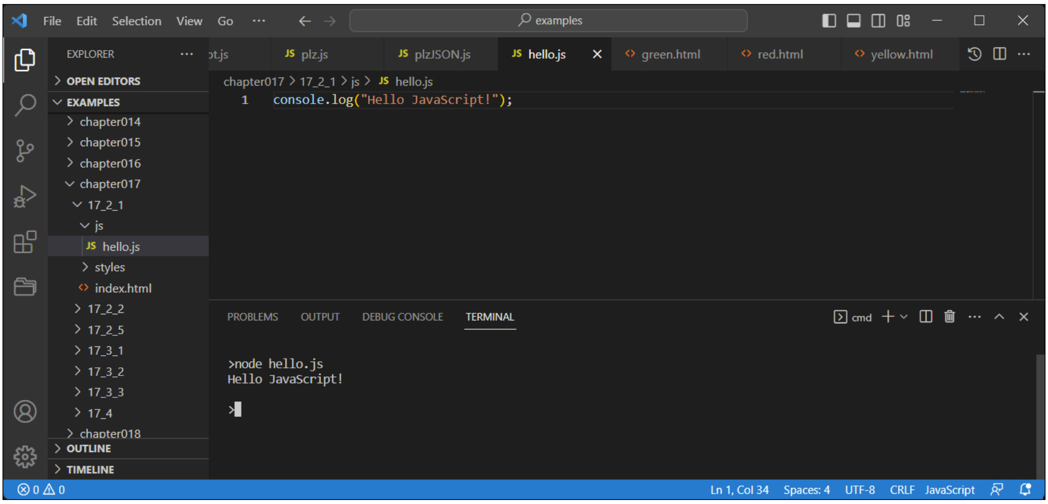
```
Your environment has been set up for using Node.js 14.15.4 (x64) and npm.

C:\Users\wolf1>node helloConsole.js
Hello JavaScript Console!

C:\Users\wolf1>
```

Figure 17.13 Node.js Allows You to Run JavaScripts without the Web Browser

Prerequisite for this example: The file name is *script.js*, and you're in the JavaScript *script.js* directory in the command line. Of course, it can even be more convenient if you use Node.js in conjunction with Visual Studio Code, where you can find the terminal and the code editor right under the same hood.

No DOM with Node.js

Logically, there is no DOM available with Node.js either because unlike JavaScript in the web browser, you aren't dealing with a web page here.


File
Edit
Selection
View
Go
examples
EXPLORER
OPEN EDITORS
EXAMPLES
chapter014
chapter015
chapter016
chapter017
17_2_1
js
hello.js
styles
index.html
17_2_2
17_2_5
17_3_1
17_3_2
17_3_3
17_4
chapter018
OUTLINE
TIMELINE
plz.js
plzJSON.js
hello.js
green.html
red.html
yellow.html
chapter017 > 17_2_1 > js > hello.js
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
cmd
Ln 1, Col 34
Spaces: 4
UTF-8
CRLF
JavaScript

```
console.log("Hello JavaScript!");
```

```
>node hello.js
Hello JavaScript!

>
```

Figure 17.14 The Perfect Duo for Writing JavaScript: Node.js + Visual Studio Code

17.3.5 Annotating JavaScript Code with Comments

In practice, it can be useful and sometimes necessary to add comments to the JavaScript code or to comment out a JavaScript code completely. You have the option to write a one-line or a multiline comment.

A one-line comment must be introduced with //. Everything after the // characters up to the end of the line will be ignored and used as a comment. This is also true if you write a JavaScript statement after //:

```
// I am a comment
// console.log('I am commented out');
```

If your comment spans multiple lines or you want to comment out a multiline block of JavaScript code, you need to introduce that multiline comment with /* and end it with */. Let's take a look at an example:

```
/*
  This is a comment,
  which can spreads across
  several lines.
*/
```

17.4 Using Variables in JavaScript

As is the case in almost any other programming language, JavaScript allows you to create and use variables. You need variables if you want to further process values or data in your script that have been entered by a user via an HTML form or read from a database, for example. Such a variable has a fixed memory address in the memory, which the JavaScript interpreter can access when needed.

JavaScript isn't a statically typed programming language, but a weakly typed or dynamic programming language, so such a variable can be of any type such as a string (string, text) or a number (integer, floating-point number). A variable can also store more complex forms of types and be an array (field of certain types) or even an object.

Term Definition: Statement(s)

A statement is almost any line of a script that ends with a semicolon. Consequently, statements are also the declaration and initialization of variables or the calling of functions.

In JavaScript, variables can be declared using either the `let` or `var` keyword. You can also initialize a variable name with values right at the declaration by using the = character. Such a variable initialization is similar to what happens in algebra:

```
let width = 5;              // Number
let pi = 3.14;              // Number
let aText = "Message";      // String
let userName = 'John Doe';  // String
let bigNum = 123456789;     // Number
```

Instead of the keyword `let` you can also use `var` here. You'll get to know the difference when we come to the scope of variables. Generally, however, you should use the keyword `let` for the definition of variables.

The variable name (also called identifier) can be almost any name. However, it must begin with a letter and mustn't contain any spaces or special characters. The only special character you can use is the underscore at the beginning or inside the variable identifier. The $ character is theoretically allowed here at the beginning or within the name. In addition, you mustn't use JavaScript keywords as variable names. It's also important to know that there's a distinction between uppercase and lowercase. With `var01` and `Var01`, you have two different variables (variable names).

Table 17.1 contains a list of JavaScript keywords; these can't be used as variable names.

JavaScript Keywords					
async	await	break	case	class	catch
const	continue	debugger	default	delete	do
else	enum	export	extends	finally	for

Table 17.1 Reserved Keywords in JavaScript

JavaScript Keywords					
`function`	`if`	`implements`	`import`	`in`	`interface`
`instanceof`	`let`	`new`	`package`	`private`	`protected`
`public`	`return`	`static`	`super`	`switch`	`this`
`throw`	`try`	`typeof`	`var`	`void`	`while`
`with`	`yield`				

Table 17.1 Reserved Keywords in JavaScript (Cont.)

As mentioned at the beginning, unlike strictly typed programming languages such as C++ or Java, JavaScript doesn't require a type to be specified. JavaScript determines the type dynamically when a value is assigned to the variable. Although you should avoid it, it's theoretically possible to change the type at program runtime.

Everything you put between single or double quotation marks will be recognized by the interpreter as a string (text), for example:

```
let aText01 = "I am a string.";      // String
let aText02 = 'I am also a string.'; // String
```

If, on the other hand, you assign a numeric value to a variable name without enclosing this value in single and double quotation marks, the interpreter will recognize this as a numeric value:

```
let width = 5;              // Number
let pi = 3.14;              // Number
let textnumber = "12345";   // Caution! String
```

Terminating Statements with a Semicolon

In JavaScript, each statement is terminated with a semicolon. If you don't use a semicolon at the end of a statement, and the line break is located at this position, JavaScript tries to insert the semicolon itself. Although you can omit the semicolon at the end of a statement, it's recommended to end even individual commands or a sequence of commands with a semicolon. This also applies when you assign values to a variable name.

Because there's frequent talk about initializing and assigning values, we'll briefly explain what this means here. You can create a variable with the keyword `let` as follows:

```
let myname;              // Agreement on a variable
console.log(myname);     // Output: undefined
```

Such an empty agreed variable without an assigned value has a value called `undefined`. After creating an empty variable, you can assign a value to it at any time using the assignment operator (again):

```
let myname;
myname = "Sample name"; // Value assignment
console.log(myname);    // Output: Sample name
```

You can initialize a variable with a value along with the agreement. Such a variable initialization looks as follows:

```
let myname = "Sample name";
console.log(myname); // Output: Sample name
```

Similarly, you can arrange more than one variable at the same time or in one statement separated by commas:

```
let myname, myfname, myage;
```

Of course, you can do the initialization right here when agreeing on multiple variables:

```
let myname = "Doe", myfname = "John", myage = 40;
console.log(myname + "," + myfname + "," + myage); // Output: 'Doe, John',
```

Likewise, once you've assigned a value to a variable, you can assign a new value to it. In the following example, `myname` first receives the value "Doe" and is then assigned the value "Deer":

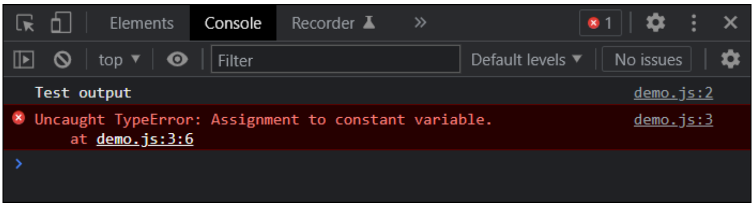
```
let myname = "Doe";
console.log(myname); // Output: Doe
myname = "Deer";
console.log(myname); // Output: Deer
```

17.4.1 Defining Constants

JavaScript also allows you to define constants. The keyword `const` is available for this purpose. You can't change the value of such a constant after initialization. In practice, constants are usually written in capital letters at the beginning of the code, for example:

```
const TVAL = 'Test output'; // Create constant
console.log(TVAL);          // Output: Test output
TVAL = 'New test output';   // Error! The order can no longer be changed.
console.log(TVAL);
```

If you try to change the value of a constant, the web console usually displays an error message. However, the behavior also depends on the runtime environment. Some of them simply ignore this assignment without throwing an error.



```
Test output                                        demo.js:2
Uncaught TypeError: Assignment to constant variable.   demo.js:3
    at demo.js:3:6
```

Figure 17.15 Google Chrome Returns an Error Message in the Console When Trying to Change a Constant Variable

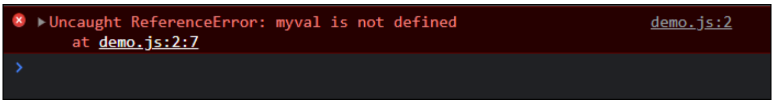
17.4.2 Strict Mode Using “"use strict"”

A JavaScript is executed in default mode without any further precautions. You can use a strict mode where certain restrictions exist. This sounds a bit negative at first, but it’s actually pretty useful because JavaScript behaves much more strictly in this mode than in standard mode. Some constructs that can be executed without problems in standard mode will result in an error in strict mode. For example, erroneous or problematic code that’s accepted in standard mode will result in an error message in strict mode. Outdated JavaScript language constructs also trigger an error message. When using strict mode, it’s sufficient to write the following statement at the beginning of the JavaScript program:

```
"use strict";
```

The following example reports an error because there’s no `let`, `var`, or `const` in front of the variable `myval`, which isn’t necessarily an error, but it implicitly creates a global variable, which you shouldn’t do:

```
"use strict";
myval = "A text"; // Error in strict mode
console.log(myval);
```



```
▸Uncaught ReferenceError: myval is not defined     demo.js:2
    at demo.js:2:7
```

Figure 17.16 Thanks to Strict Mode, the JavaScript Reports an Error Here

The problem with a global variable without `let` is that this variable is implicitly defined as a property of the global object, which is, for example, the `window` object in the web browser. Such global variables could override properties of the global object.

The strict mode makes sure that the use of error-prone features of JavaScript is simply not allowed. This leads to a script abort for previously ignored errors (without `"use strict"`).

17.5 Overview of JavaScript Data Types

As you already know, unlike other languages such as C++ or Java, you don't have to specify a data type when declaring variables because this is determined at runtime in JavaScript based on the value that has been passed.

JavaScript defines multiple data types. The primitive types are `string`, `number`, `boolean`, and `symbol`, and the special types include `undefined` and `null`. In addition to the primitive data types, JavaScript also has the composite data type `object` for objects.

You can determine the type of a variable using the `typeof` operator. Possible return values are `string`, `number`, `boolean`, `object`, `function`, `symbol`, and `undefined`.

17.5.1 Number Data Type (Numbers)

In JavaScript, there's no difference between integers and floating-point numbers. According to the ECMAScript standard, there's no specific data type for integers, and all data types for numbers are represented internally by JavaScript as 64-bit floating-point values. For example:

```
let integerValue = 12345;
console.log(typeof integerValue); // Output: number
let floatingPointValue = 123.123;
console.log(typeof floatingPointValue); // Output: number
```

If a value doesn't correspond to a correct numerical value, `NaN` (`NaN` = *not a number*) will be used as the value. If the value range has been exceeded or fallen below, `Infinity` or `-Infinity` will be used as the value. For this reason, there are two constants: `Number.POSITIVE_INFINITY` and `Number.NEGATIVE_INFINITY`. If you want to determine the smallest or largest possible number you can use, the `Number.MIN_VALUE` and `Number.MAX_VALUE` constants are useful:

```
console.log(Number.MIN_VALUE); // Output: 5e-324
console.log(Number.MAX_VALUE); // Output: 1.7976931348623157e+308
console.log(Number.NEGATIVE_INFINITY);  // Output: -Infinity
console.log(Number.POSITIVE_INFINITY);  // Output: Infinity
```

When specifying floating-point values, you must use a period instead of a comma. For higher or smaller floating-point values, you can use the E notation. For example, a specification of `5e-3` corresponds to 0.005. With -3, the decimal point is shifted to the left by as many digits as are indicated after the E character. The same is true for `54321e3`, which moves the decimal point three places to the right because the number after E is positive. Thus, `1.2e4` corresponds to the value 12,000:

```
let floatingPointValue1 = 5e-3;
console.log(floatingPointValue1); // Output: 0.005
let floatingPointValue2 = 1.2e4;
console.log(floatingPointValue2); // Output: 12000
```

17.5.2 String Data Types (Strings)

Strings are used to represent text and consist of a string of zero or more 16-bit characters according to the UCS-2 encoding for letters, digits, and punctuation marks. You can insert such string literals in JavaScript by placing a text between single or double quotes. In JavaScript, there are no primitive data types for a single character as there are in other programming languages.

Let's take a look at a simple example:

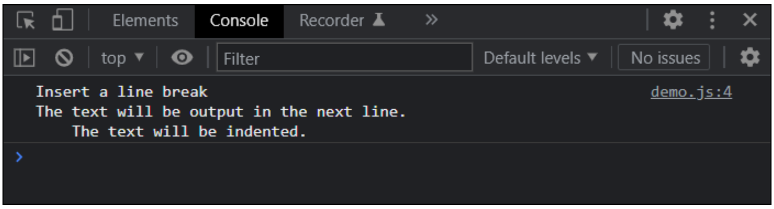
```
let aText1 = "String in JavaScript";
console.log(typeof aText1); // Output: string
let aText2 = 'Also a string in JavaScript';
console.log(typeof aText2); // Output: string
let aText3 = "12345";
console.log(typeof aText3); // Output: string
```

Whether you use single or double quotation marks is up to you. A good style is to choose one version and then use it consistently. What's more convenient about using single quotes is that you can use special characters within the string without an escape sequence, which is an advantage especially because double quotes are often used in some countries. For example:

```
let aText4 = "Quotation marks in \"text\""; // escape sequence needed
console.log(aText4); // Output: Quotation marks in "text"
let aText5 = 'Quotation marks in "text"'; // without escape sequence
console.log(aText5); // Output: Quotation marks in "text"
```

Escape sequences are control characters that you can insert into strings as variable values. Such control characters are preceded by the character `\` followed by the letter marking the control character. For example, you can insert a line break using the control character `\n`, or you can insert a tab feed via `\t`:

```
let aText6 = "Insert a line break\n"; // line break at the end
let aText7 = "The text will be output in the next line.\n";
let aText8 = "\tThe text will be indented.\n";
console.log(aText6 + aText7 + aText8);
```



```
Insert a line break
The text will be output in the next line.
    The text will be indented.
```

Figure 17.17 The Example with a Line Break and a Tab Feed during Execution

The most important control characters that are relevant in JavaScript applications are listed in Table 17.2.

Control Characters	Meaning
`\'`	Outputs a single quote character inside the string.
`\"`	Outputs a double quote character inside the string.
`\\`	Outputs the backslash character inside the string.
`\n`	Outputs a line break. However, the line break applies only to the console or the standard JavaScript tip dialogs. For a line break on web pages, of course, you use `<br>` instead of `\n`.
`\t`	Outputs the tab character, which means an indentation to the right. This control character also affects only the console or a standard message dialog.
`\uXXX`	This enables you to add a Unicode symbol. For this purpose, hexadecimal values are used. A specification such as `\u00A9` adds the © character to the string.

Table 17.2 Control Characters for Strings

Term Definition: Literals

One term you may come across more often is "literal," which simply refers to the value that's fixed in the code behind a string or a number. Values such as `56`, `"Doe"`, `3.1415`, `'Are you sure?"`, or `2016` represent literals that literally occur as shown in the code:

```
let year = 2016; // 2016 is a literal.
let name = "Doe"; // "Doe" is a literal.
let query = 'Are you sure?'; // 'Are you sure?' is a literal.
let pi = 3.1415; // 3.1415 is a literal.
```

If you want to link individual strings together, you can do this using the + operator:

```
let text1 = "to be ";
let text2 = "not ";
let text3 = text1 + "or " + text2 + text1;
console.log(text3); // Output: to be or not to be

let text4 = "The value is: " + 123.123 + 100;
console.log(text4); // Output: The value is: 123.123100
```

When you mix numbers and strings with the + operator, the result will always be a string, as you can see in the example with `"The value is: "` + `123,123` + `100`.

17.5.3 Template Strings

Template strings are string symbols that may span multiple lines and also allow embedded JavaScript expressions. Such template strings are enclosed between two grave accents (`) instead of between double or single quotes. This allows you to do without line breaks using the `\n` sequence. For example:

```
let text1 = "Last line";
console.log("First line\n" + "Second line\n" + text1);
// First line
// Second line
// Last line
```

You can write the same with the template strings as follows:

```
...
console.log(`First line
Second line
${text1}
`);
```

I think this looks much clearer because you can enter the text as you want it to be displayed. To insert JavaScript expressions into the strings, you can use the notation `${expression}`. In the example, the value of the variable `text1` was used as an expression.

17.5.4 Boolean Data Type

The Boolean data type can have only two values with the literals `true` and `false`. A Boolean value is a truth value and usually expresses the validity of a condition or expression. In addition, conditions that contain the value `0`, an empty string, `NaN`, `undefined`, or `null` are interpreted as `false`. All other values are `true`. There's nothing more to say about this at the moment. Nevertheless, here are a few simple string and number comparisons for testing:

```
let val1 = 123;
let val2 = 234;

let isAdmin = false;
let isUser = true;

console.log(val1 > val2); // Output: false
console.log(val1 < val2); // Output: true
```

You'll use the Boolean values even more often in practice when it comes to branching within a JavaScript.

17.5.5 Undefined and Null Data Types

A variable that hasn't yet been assigned a value has the value `undefined`. In addition, a nonexistent object property or nonexistent function parameters have the value `undefined`. For example:

```
let myname;
console.log(name); // Output: undefined
myname = "Wolfe";
console.log(myname); // Output: Wolfe
```

With the datatype `null`, on the other hand, you represent an empty object. In the following example, I anticipate the *objects* theme and create a `mname` object from last name and first name. I'll then assign the value `null` to the `mname` object, which deletes the contents of the object. This specifies that the object variable hasn't been assigned any values, that is, it's an empty object.

```
let mname = {
    lname: 'Wolfe',
    fname: 'Jason'
};
console.log(mname); // Output: {lname: "Wolfe", fname: "Jason"}
mname = null;
console.log(mname); // Output: null
```

Unlike `undefined`, `null` is a JavaScript keyword. The type `null` was listed as a data type, but a `typeof null` returns the type `object`. Variables you've initialized with `null` are thus of the type `object`. An uninitialized variable, on the other hand, is `undefined`. Unlike `null`, `undefined` isn't meant to be assigned to a variable. The value `undefined` simply indicates that a variable hasn't yet been initialized with a value. `null`, on the other hand, is an empty object.

17.5.6 Objects

Objects are a collection of properties and methods, where a *method* is a function, and a *property* is a value or set of values of an object. In addition to browser objects and predefined objects, you can also create and use your own objects in JavaScript. The topic is quite important in JavaScript, so we won't just cover it here in a subsection. The objects would also come too early at this point because you still need some basics in JavaScript itself for that. You can learn more about objects in JavaScript in Chapter 18.

17.5.7 Converting Data Types

Because JavaScript is very flexible and dynamic when it comes to data types, and because you can convert data types automatically during execution, I'll get a little more specific about type conversion here. The following example shows what is meant by dynamic typing:

```
let val = 123;
console.log(typeof val); // Output: number
val = "Now a string";
console.log(typeof val); // Output: string
```

While such an example isn't intended to form a precedent, it shows how the variable `val` is first used as a numerical value, and after it was assigned a string, the data type got converted into a string.

You've already seen something similar when you added a string with a numerical value using the + operator, whereupon the number was converted to a string:

```
let text = 5 + " You should be friends";
console.log(typeof text); // Output: string
```

However, this only applies to the + operator in conjunction with strings, which is used to link them together. On the other hand, if you use other operators such as -, *, or /, JavaScript no longer converts the numbers to strings, but tries to convert the strings to numbers. Consider this example:

```
let text1 = "100" - 44;     // 56
console.log(typeof text1); // Output: number
let text2 = "100" + 44;     // "10044"
console.log(typeof text2); // Output: string
console.log("10" / "2");    // Output: 5
```

For `"100" - 44`, the string `"100"` is converted to the number 100. The reason is that the arithmetic minus operator expects two numbers as operands. The same would happen if you were to calculate `"100" - "44"`. Both strings would be implicitly converted to numbers. You can see the example here with `"10" / "2"`. With `"100" + 44`, however, a string

`"10044"` would be created because this is the standard behavior when the `+` operator is used in connection with strings.

After all, JavaScript can't know which data type you need at any given moment. If you want to perform an addition operation on an example such as “`100`” `+ 44`, you must explicitly convert the string “`100`” to a number. For such purposes, JavaScript provides the functions `parseInt()` and `parseFloat()`. You can use `parseInt()` to convert a string to an integer and `parseFloat()` to convert a string to a floating-point number. Here's an example with `parseInt()`:

```
let iVal = parseInt("100") + 44; // 144
console.log(typeof iVal);        // Output: number
```

In this example, the string `"100"` was first converted to an integer with `parseInt()`. Because the two operands to the right and left of the + operator are numbers, an addition is performed, and a number is passed to `iVal` as the result. For such conversions from a string to a number, you must check that the conversion was done properly and that the number isn't `NaN` afterwards. Often the data doesn't come as a simple literal as in the examples, but is entered by a user or read from a database. For this purpose, you can use, for example, the function `isNaN()`, which returns `true` if the number is invalid. Otherwise, it returns `false`. Let's take a look at the following example:

```
let notANumber = "100 elements" - 50;
console.log(typeof notANumber);    // Output: number
console.log(notANumber);           // Output: NaN
console.log( isNaN(notANumber) ); // Output: true
```

The arithmetic calculation of `"100 elements" - 50` doesn't provide a meaningful value. Although `typeof` returns `number` here, the output of the variable `notANumber` confirms that no valid value was calculated and stored here, which is why the result is `NaN`. The check using the `isNaN()` function confirms this too. The `isNaN()` function is therefore needed because it isn't possible to check the value for `NaN` with `==`.

17.6 Arithmetic Operators for Calculation Tasks in JavaScript

Like any other programming language, JavaScript has all the common arithmetic operators for numerical calculations on board. Table 17.3 provides an overview of the arithmetic operators in JavaScript.

Operator	Meaning	Example
+	Addition	`a = b + c;`
-	Subtraction	`a = b - c;`

Table 17.3 Overview of the Arithmetic Operators

Operator	Meaning	Example
*	Multiplication	a = b * c;
/	Division	a = b / c;
%	Remainder of a division	a = b % c;

Table 17.3 Overview of the Arithmetic Operators (Cont.)

The use of arithmetic operators is relatively simple, as the following example shows:

```
let val1 = 101 + 202;
console.log(val1);      // Output:  303
console.log(88 - 22);  // Output:  66
let val2 = val1 * 3;
console.log(val2);      // Output: 909
console.log(val2 / 4); // Output: 227.25
console.log(val2 % 4); // Output: 1
```

As usual in mathematics, when multiple operators are used, the rule applies that multiplication and division tasks are done before addition and subtraction tasks. Thus, in an expression such as 15 – 2 * 5, 2 is first multiplied by 5, then the result of the expression 2 * 5 is subtracted from 15, resulting in 5. Thus, the highest priority arithmetic operators are *, /, and %. Only then do + and – follow and finally the assignment operator =. Except for the assignment operator, arithmetic operators of the same rank are evaluated from left to right.

Consider this example:

```
let val3 = 100 / 2 - 5 * 4;
console.log(val3); // Output: 30
```

In this example, the subexpressions `100 / 2` (= 50) and `5 * 4` (= 20) are calculated first, and then the result of these subexpressions is subtracted (50 – 20), resulting in 30. If you don't want to perform a calculation according the preceding rule, you can use parentheses. The use of parentheses has the highest priority and, if nested, gets evaluated from the inside out. Here's an example:

```
let val4 = 5 + 6 * 2;      // = 17
let val5 = (5 + 6) * 2;    // = 22
```

In the first example, 6 * 2 (= 12) is first calculated as usual and then 5 is added, which leads to the result 17. In the second example because of the higher priority of parentheses over operators, the expression between the parentheses, 5 + 6, is calculated first (= 11) and then multiplied by 2 (= 22). Often the use of parentheses is helpful because it makes the code more readable. For example, a calculation such as (100 / 2) - (5 * 4) reads better than 100 / 2 – 5 * 4.

Furthermore, in JavaScript, in addition to the ordinary assignment operator =, you can find arithmetic compound assignment operators such as +=, -=, *=, /=, and %=. Again, the meaning is the same as listed in Table 17.3. With the arithmetic assignment operators, instead of a calculation like `valA=valA+valB;`, you can just write `valA+=valB;` quickly and briefly. The same applies to the other versions. However, when using compound assignment operators, there must be a variable on the left. In terms of priority, these operators are on the same level as the assignment operator.

Mathematical Functions

In JavaScript, there's a `Math` object that allows you to use various mathematical functions. In general, you should use `Math` if you need mathematical calculations without rounding errors. For example, `Math.random()` returns a random number. There are also more complex mathematical methods such as `Math.sqrt(x)`, which returns the square root of `x`.

17.7 Conditional Statements in JavaScript

Conditional statements or branches allow you to influence the flow of the program by defining a condition and thus deciding at which point the program should be continued. The following options are available to you:

- You can use `if` to branch to a block of statements that are executed only if the condition in the parentheses of `if()` equals `true`. This is also called a conditional branch.
- You can use `else` with a block of statements that will be executed only if the previously checked `if` condition was equal to `false`.
- `else if` can be used with a block of statements to test another condition if the preceding `if` was equal to `false`. After a preceding `if`, you can use multiple `else if` conditions. Alternatively, you can use `switch()` for such a multiple branch.

In your daily work, you'll probably use conditional branching with **`if`** and the alternative **`else`** most often. Here's the syntax of such an **`if`-`else`** construct:

```
if ( condition==true ) {
  // statements if condition is true
}
else {
  // statements if condition is false
}
```

As a condition itself, you can use any expression that can be evaluated to a Boolean truth value. I'll describe the Boolean truth value in the next section. At this point, it should be noted that the alternative `else` branch is optional here.

Statement Blocks

You already know a block with statements (or statement block) from the functions; that is, several statements are combined in one block. Such a block starts with an opening curly bracket ({) and ends with a closing curly bracket (}). JavaScript doesn't require a semicolon at the end of the curly bracket.

17.7.1 "true" or "false": Boolean Truth Value

A Boolean truth value is specified as `true` or `false` in JavaScript. Simply put, it can be said that anything containing a true value equals `true`, and anything without a true value is `false`. Nevertheless, JavaScript isn't limited to the Boolean values `true` and `false`. There are also values that are considered false (also called falsy), such as `undefined`, `null`, `0`, or `""`. Such falsy values are treated as `false`. Other values that aren't falsy, on the other hand, are considered true and are truthy. Thus, objects (without properties), functions, or arrays (with length `0`) are truthy. This means that `""` is equal to falsy, and `" "` is truthy. Arrays are described separately in Chapter 18, Section 18.2.

Real values and therefore `true` are the following examples:

```
1234
1.234
-1
"A text"
5 + 1 * 2
```

Here's an example for demonstration purposes:

```
let mytext = 'A text';
if (mytext) {
    console.log('"mytext" is a valid value.'); // <- Output
} else {
    console.log('"mytext" is an invalid value.');
}
```

Here, the `if` condition equals `true`, so `mytext` is a valid value, which is why the corresponding output is executed in the curly brackets following it.

The following examples have no real values and therefore are always `false`:

```
0             // The number 0 is false.
""            // An empty string is false.
```

```
var val01; // Empty variable is undefined and therefore false.
var val02 = false;
100 / "text" // is NaN (= Not a Number), therefore false
null         // null is always false.
NaN          // Not a Number, no number is false.
```

Here's another example:

```
let mytext = "A text";
let val01 = 100;
if (val01 / mytext) {
    console.log('Calculation successful');
} else {
    console.log('NaN -> no valid value.');
}
```

Here the statements would be executed in the alternate `else` block because the `if` condition returns `false`. The division of `100 / "A text"` results in the symbolic value `NaN` (`NaN` = *not a number*) and is therefore invalid and `false`.

17.7.2 Using the Various Comparison Operators in JavaScript

Besides the possibility to check whether a value is valid and equals `true` or just an invalid value and thus returns `false`, you can compare variables and values using the various comparison operators. Depending on whether the comparison is true or false, `true` or `false` will be returned here as well. To perform comparisons, JavaScript provides the comparison operators listed in Table 17.4.

Operator	Description	Example (x=6; y=5)
`==`	Same as	`x==5; // false`
`!=`	Unequal to	`x != 5; // true`

`===`	Same value and type	`x === y; // false`
		`x === 6; // true`

`!==`	Different value or different type	`x !== y; // true`
		`x !== 6; // false`

`>`	Greater than	`x > y; // true`
`<`	Less than	`x < y; // false`

`>=`	Greater than or equal to	`x >= y; // true`
		`x >= 6; // false`

Table 17.4 Comparison Operators in JavaScript

Operator	Description	Example (x=6; y=5)

`<=`	Less than or equal to	`x <= y; // false`
		`x <=6; // true`

Table 17.4 Comparison Operators in JavaScript (Cont.)

17.7.3 Using the “if” Branch

With the background knowledge of Boolean truth values and the comparison operators, you’ll be able to apply the `if` branches in practice. Let’s take a look at a simple example:

```
let age = prompt('How old are you: ');
if (age >= 18) {
    console.log("Access granted")
} else {
    console.log("Access denied");
}
```

When you run the example, the `prompt()` method opens a dialog in the browser window with an input field and an **OK** and **Cancel** button. The value you enter here in the input field is returned by `prompt()` and assigned to the variable `age` in the example. The `if` statement then checks whether the value of the variable `age` is greater than or equal to 18. If this condition is `true`, a corresponding output will appear in the console. If the condition is `false`, the statement is executed in the alternate `else` branch.

At this point, I’d like to share a few words about the comparison operators `===` and `!==`. These are necessary because with comparison operators, an *implicit type conversion* is performed before the comparison so that these values can be compared. In the following example, a string is compared to an integer:

```
let strVal = "1234";   // number
let iVal = 1234;       // string
if (strVal == iVal) { // true because type conversion
    console.log("Both values are equal");
} else {
    console.log("Values are different")
}
```

In this example, the comparison of `"1234"==1234` returns `true` due to a type conversion. For such purposes, the operators `===` and `!==` are available, which compare not only the value but also the type. So, if you replace the `==` operator with the `===` operator in the example, `true` will no longer be returned because the value is the same but the type isn’t.

17.7.4 Using the Selection Operator

You can shorten an `if-else` construct with the selection operator. This is very useful, for example, if you want to assign a specific value to a variable depending on the condition. The structure of the operator is as follows:

```
let val = condition ? value1 : value2;
```

Here the `value1` value is assigned to the variable `val` if `condition` is `true`. If `condition` is `false`, the `value2` will be assigned to `val`. With regard to the `if-else` construct, this roughly corresponds to the following:

```
let val;
if (condition) {
  val = value1;
} else {
  val = value2;
}
```

The `else` branch can be omitted if `val` is initialized with `false` at the beginning. However, this is also more about demonstrating the counterpart of the selection operator.

Here's an example of the selection operator, where `prompt()` is used to query a pseudo password, and according to the input, `true` or `false` gets assigned to the `isAdmin` variable:

```
let pwd = prompt('Enter password: ');
let isAdmin = pwd == 12345678 ? true : false;
console.log(isAdmin);
```

17.7.5 Logical Operators

The logical operators in JavaScript are `&&` (AND), `||` (OR), and `!` (NOT). Logical operators are used with truth values. If you use numbers, they will be implicitly converted to a truth value before they are linked with `&&`, `||`, or `!`.

Operator	Meaning
`&&`	Expressions linked with the AND operator return `true` only if all expressions are `true`. Example:

```
if ( ival1 > 0 && ival2 > 0 ) {
  // Both expressions are true = true.
}
```

Table 17.5 The Logical Operators in JavaScript

Operator	Meaning
`||`	Expressions combined with the logical `OR` operator return `true` if at least one of the expressions is true. Example:

	`if ( ival1 > 0 || ival2 > 0 ) {`
	`  // At least one expression is true = true.`
	`}`

`!`	You can use the logical `NOT` operator to negate an expression. So, you can turn “true” into “false” and vice versa. Example:

	`if( !(ival > 0) ) {`
	`  console.log("ival is not greater than 0");`
	`}`

Table 17.5 The Logical Operators in JavaScript (Cont.)

Here’s a simple example in which you’re again supposed to enter a value between 1 and 100 via the prompt of a `prompt()` dialog:

```
let val = prompt('Enter a value from 1 - 100: ');
if (val >= 1 && val <= 100) {
    console.log("The value matches the requirements.")
} else {
    console.log("Wrong input: " + val);
}
```

Using the logical `AND`, the expressions of whether the entered value `val` is equal to or greater than 1 `AND` equal to or less than 100 have been linked here, and `true` will only be returned if both expressions are `true`.

If you had used the logical `OR` operator `||` instead, `true` would always be returned if the entered value is greater than or equal to 1 because the second expression wouldn’t have been checked at all. However, if the value is negative or 0, the `OR` operator in the example also evaluates the second expression. In the case of a link with the logical `OR`, the check aborts at the first `true` because the condition is that at least one link is true.

Linking More Expressions Together

Of course, you can link more than two expressions together or mix the `&&` and `||` operators. However, you should keep in mind the readability of the source code.

17.7.6 Multiple Branching via “switch”

If you want to check multiple cases, you can theoretically use multiple `if` queries in a row, or you can use the case distinction, `switch`. In JavaScript, `switch` supports values with any type. It’s even possible to have values determined dynamically first via function calls. In many other programming languages, these values must be constants. For this purpose, here's a theoretical construct of a `switch` case distinction:

```
switch( expression ) {
  case label_1:
    statements;
    [break];
  case label_2:
    statements;
    [break];
...
  default:
    statements;
}
```

In this case distinction, a matching value in `case` is searched for the expression in `switch`. If a `case` marker matches the `switch` evaluation, the program execution takes place after this `case` marker. If no `case` marker matches the `switch` evaluation, you can use an optional `default` marker, which will be executed as an alternative. Of particular importance in a `switch` case distinction are the `break` statements at the end of a `case` marker. You can use `break` to instruct the program to jump out of the `switch` block and continue with the program execution after. If you don't use a `break` statement, all further statements (including the `case` markers after them) in the `switch` block will be executed until the next `break` statement or the end of the block has been reached.

Here's a simple example to demonstrate `switch` in use:

```
switch (new Date().getDay()) {
    case 0:
        console.log("Today is Sunday");
        break;
    case 6:
        console.log("Today is Saturday");
        break;
    default:
        console.log("Today is an ordinary weekday");
}
```

Here, a `Date` object is created in switch, which is why the `getDay()` method is called. The `getDay()` method returns a day of the week as a number. 0 is returned for Sunday, 1 for Monday, and so on to 6 for Saturday. In the example, the return value is compared with `case` markers 0 (for Sunday) and 6 (for Saturday). If one of the markers matches, a corresponding output will occur in the JavaScript console. If none of the `case` markers apply, the return value is 1, 2, 3, 4, or 5, and it's a normal weekday, so that no `case` marker will be used here anymore but `default`.

17.8 Multiple Repetitions of Program Statements via Loops

Loops are very suitable if you want to repeat certain statements multiple times. JavaScript supports several types of loops, which I'll briefly describe in the following sections.

17.8.1 Increment and Decrement Operators

An increment or decrement increases or decreases the value of a variable by 1. These two operators are predominantly used with loops. The operators are written in JavaScript as follows:

Operator	Meaning
`++`	Increment operator; variable is incremented by 1.
`--`	Decrement operator; variable is decremented by 1.

Table 17.6 Increment and Decrement Operators

There are two possibilities for the use of these two operators: postfix notation and prefix notation.

Usage	Meaning
`val++`	Postfix notation; increments the value of `val`, still passes the old value to the current expression
`++val`	Prefix notation; increments the value of `val` and passes it immediately to the current expression
`val--`	Postfix notation; reduces the value of `val`, still passes the old value to the current expression
`--val`	Prefix notation; reduces the value of `val` and passes it immediately to the current expression

Table 17.7 Postfix and Prefix Notations

Here's a simple example to demonstrate the increment operator (`++`) in more detail. The same applies to the decrement operator (`--`).

```
let iVal = 1;
console.log("iVal = " + iVal);   // Output: iVal = 1
iVal++;
console.log("iVal = " + iVal);   // Output: iVal = 2
console.log("iVal = " + iVal++); // Output: iVal = 2
console.log("iVal = " + iVal);   // Output: iVal = 3
console.log("iVal = " + ++iVal); // Output: iVal = 4
```

The first time the increment operator is used, the old value is still passed to the current expression. Because the increment is standing alone here, this is also the current expression. The next line of output, on the other hand, is the next expression, which is why the value of `iVal` is `2` in this case. You'll understand it better if you run `iVal++` inside `console.log()` in the next line. Here the output is still `2` because the increment is executed within the current expression. The current expression ends at the semicolon where the late increment (and, if used, decrement) takes effect. The next expression in the next line has the expected value `3`. If you want to increment the value of a variable immediately within an expression, you must use the prefix notation instead of the postfix notation, as I did in the last line with `++iVal`.

17.8.2 The Header-Controlled "for" Loop

You'll probably want to use the flexible `for` loop most often. The syntax for this loop looks as follows:

```
for (initialization; condition; increment/decrement) {
  // statement block that will be executed
}
```

`Initialization` is executed only once when the loop is started and is usually used to set a count variable for the loop. `Condition`, on the other hand, usually represents the condition for the statements in the statement block of the loop to be executed. As long as the condition in `Condition` equals `true`, the loop will be executed again. If `false`, the loop terminates, and program execution continues after the statement block of the `for` loop. As a condition, it's often checked whether the count variable corresponds to a certain value. `Increment/Decrement`, on the other hand, is always executed when the statements in the statement block have been executed. Most of the time, you want to change the count variable of the loop here.

A simple use of the `for` loop could look as follows:

```
for (let i = 0; i < 3; i++) {
    console.log(i + 1 + "-th loop pass");
}
```

For demonstration purposes, the loop was output to the console. In the process, the statement was executed three times in the loop. The count variable `i` was first set to 0, the condition `i<3` was checked and then the statement block after it was executed. The output in the JavaScript console shows how many times the loop has been executed. Next, the loop variable is incremented by 1 with `i++`, and the condition `i<3` is checked again, which (`i=1`) is still `true`. The process gets repeated until the value of `i` equals `3` and thus the condition `i < 3` returns `false`.

The output of the example in the console looks as follows:

```
1st loop pass
2nd loop pass
3rd loop pass
```

All Three Expressions in the "for" Loop Are Optional

All three expressions in the `for` loop are optional and can be omitted. In any case, you must use the two semicolons in the `for` loop. Theoretically, `for(;;)` would be a valid `for` loop. Note, however, that if you omit the second expression, you'll create an infinite loop, which could crash the browser sooner or later. If you use an infinite loop, you should use a `break` inside this loop. Such a `break` can be used to jump out of the loop.

17.8.3 The Header-Controlled "while" Loop

The `while` loop is a header-driven loop and is executed as long as the condition in `while` returns `true`. Here's the syntax:

```
while (condition) {
  // statement block that will be executed
}
```

In practice, you can `loop` through a block of statements using the `while` loop as follows:

```
let i = 0;        // Initialize counter variable
while (i < 3) {   // Check condition
    console.log(i + 1 + "-th loop pass");
    i++;          // Increment count variable
}
```

You know the example from the `for` loop, except that here you've initialized the count variable before the loop pass, and you increment the loop variable itself at the end of the statement block. The danger in this case, in contrast to the example with the `for` loop, might be that you forget to increment the variable for the loop condition. In practice, a `while(condition)` corresponds to a `for(;condition;)`.

17.8.4 The Footer-Controlled "do-while" Loop

In the `do-while` loop, as opposed to the `while` loop, the condition isn't checked until the end, when the statement block has been executed. Thus, in the `do-while` loop, the statement block of the loop is executed at least once before the condition gets checked. Here's the syntax for the `do-while` loop:

```
do{
  // statement block that will be executed
} while (condition);
```

Here's an example in which a loop pass occurs three times:

```
let i = 0;
do {
    console.log(i + 1 + "-th loop pass");
    i++;
} while (i < 3);
```

17.8.5 Ending the Statement Block Using "break"

A statement that can be very useful within a loop is the `break` statement. If you use a simple `break` inside a statement block of the loop, it will jump out of the loop execution, and the script will continue after the loop. Here's a short code snippet:

```
let i = 0;
while (i < 10) {
    console.log(i + 1 + "-th loop pass");
    i++;
    if (i === 5) {
        console.log("End loop with break");
        break;
    }
}
```

As a matter of fact, this loop should be run 10 times according to the condition in `while`. However, this loop runs only 5 times because then the `if` condition `i==5` returns `true` so that the `break` statement ensures that the loop terminates prematurely.

17.8.6 Jumping to the Start of the Loop via "continue"

With `continue`, you end a loop pass and jump back to the beginning of the loop. This is useful if you don't want to execute the further loop pass due to a condition. Let's take a look at a simple example:

```
let i = 0;
while (i < 10) {
    i++;
    if (i % 2 === 1) {
        continue;
    }
    console.log("Value divisible by 2: " + i);
}
```

In the example, a loop is incremented from 1 to 10 and passed through. Each loop pass checks whether the value of `i % 2` (i modulo) results in a remainder. If the condition is `true`, `continue` jumps back to the beginning of the loop. If the condition is `false`, it's a number divisible by two or an even value, and the number is output.

17.9 Summary

This chapter was just a little roundabout way to introduce you to how you can use the basic JavaScript techniques. In the next chapter, you'll expand this knowledge to include objects, arrays, and functions before slowly moving on to developing JavaScript applications that run directly in the web browser. You're now familiar with the following basic JavaScript programming techniques, among others:

- Write and run JavaScript programs.
- Create an output.
- Handle variables and values.
- Work with the basic data types of JavaScript.
- Use branches and loops.

Chapter 18

Arrays, Functions, and Objects in JavaScript

Objects and object-oriented programming are particularly important in JavaScript. Objects are the main data types in JavaScript. Much of JavaScript is somehow an object. For JavaScript, a web page is virtually already an object. Arrays and functions are also essential topics that you have to deal with every day in JavaScript development.

Because you'll be dealing a lot with arrays, functions, and objects in web programming, you should at least know the basics. However, you won't find a comprehensive treatise in this book. If you're already familiar with another object-oriented language, you'll have an easier time reading into the chapter, but you'll find that much is a bit different in JavaScript.

This chapter deals with the following topics:

- **Functions**
  You'll learn how to use recurring statements in JavaScript in functions (also called subroutines) and thus be able to reuse them at any time.
- **Arrays**
  It can become quite cumbersome to always put data into a single variable of primitive data types when the data size gets larger. This is where arrays come into play, which can be used to store entire lists of different variables.
- **Objects**
  As mentioned earlier, JavaScript is a language that supports and lives the object-oriented paradigm. Even functions are objects in JavaScript. For this reason, objects must also be addressed in this chapter.

18.1 Functions in JavaScript

Functions in JavaScript are subroutines with recurring JavaScript statements that you group together in a block and call using the function name and optional arguments. Optionally, such a function returns a value to the caller of the function.

If you come from a different programming language, it may be useful to know at this point that functions in JavaScript are real objects (i.e., first-class objects) and can therefore be assigned to variables. Likewise, functions can be used as parameters or return values of functions. If JavaScript is your first programming language, you shouldn't pay much attention to this information yet, but keep it in mind.

18.1.1 Different Ways to Define a Function in JavaScript

In JavaScript, there are several ways to use functions (i.e., function objects), which I'll describe in more detail in the following sections.

Defining Functions: Function Declaration

A JavaScript function must be introduced via the keyword `function`. This is followed by an identifier (the function name) and the parentheses (`()`). The name of the function is subject to the same restrictions as the variable names. Inside the parentheses, you can optionally write a list of formal parameters. Multiple parameters must be separated with a comma. The actual code or statement, also referred to as the *function body*, must be written between curly brackets (`{}`). Here's the complete syntax of a function in JavaScript:

```
function sum(parameter1, parameter2) {
    // Code for the function
    let sum = parameter1 + parameter2;
    return sum;
}
let val1 = 200, val2 = 100;
let total = sum(val1, val2);      // Function call
console.log("Result=" + total);   // Output: Result=300
```

Listing 18.1 /examples/chapter018/18_1/script.js

The code inside the function gets executed when the function with `sum()` and the arguments (here, 200 and 100) is called. Optionally, such a function can also return a value to the caller with `return` if the caller of the function needs to continue working with that value. If you don't want to return a value from a function, you can omit the `return` statement. JavaScript doesn't require a semicolon at the end of the curly brackets of the function block.

Formal Parameters and Arguments

We refer to a formal parameter when we define the parameters in the program code. The arguments (or actual parameters) are the values we use in a function call.

Defining a Function: Function Expression

In addition to a function declaration, you can also create a function as a function expression in JavaScript. This works quite similar to the function declaration in the previous example, except that you assign the function to a variable, which makes this reference variable subsequently refer to the function object and can be used like an ordinary function:

```
let sum = function(parameter1, parameter2) {
    return parameter1 + parameter2;
}
let val1 = 100,
    val2 = 200;
let total = sum(val1, val2);    // Function call
console.log("Result=" + total); // Output: Result=300
```

Listing 18.2 /examples/chapter018/18_1/script2.js

In a function expression, you don't use a function name, which makes this function an *anonymous function*.

Constructor functions

Up until now, I haven't mentioned the more complex use of creating a function with the constructor because this is part of the programming of objects. The function is prefixed with the keyword `new`.

Use Function Declarations or Function Expressions?

For function declarations, you must give the function a name, while that's optional for function expressions. As a result, in function declarations, you can call the function using the function name. In function expressions, this is done via the variable assigned to the function.

With function declarations, the interpreter can process the function at any time, even if the function declaration is written in the JavaScript code after the function call. For example, the following code can be executed without any problem:

```
let total = sum(100, 200);  // Function call
function sum(parameter1, parameter2) {  // Function declaration
    return parameter1 + parameter2;
}
```

With function expressions, however, this isn't possible. The interpreter processes function expressions only if the corresponding statements exist; that is, the JavaScript code of the function expression must precede the function call. The following code would therefore not work with function expressions:

```
let total = sum(100, 200);  // !!! Incorrect function call !!!
let sum = function(parameter1, parameter2) {  // Function expression
    return parameter1 + parameter2;
}
```

18.1.2 Calling Functions and Function Parameters

If the function has been defined, you can call it by the function name. In the previous example, you've already done this by using `sum(val1, val2)`. Using this function call, you pass the two values of `val1` and `val2` as arguments to the function `sum()`. In the example, these two values were added together in the function, `parameter1 + parameter2`, and returned.

Parameters and Arguments

Parameters are understood to be the signature specified in the function definition. The arguments, on the other hand, are what you pass to the function as a value when you call it.

If a function expects parameters, you can group them between parentheses. The individual parameters must be separated by a comma. You must also use parentheses `()` after a function call if the function doesn't contain a parameter.

In JavaScript, it isn't an error if you call a function with fewer or more arguments than you specified parameters in the function declaration. If you've used too few arguments in the call, the missing arguments will be initialized with the default value `undefined`. For example:

```
function simpleFunc( param1 ) {
   console.log(param1);  // Output: undefined
}
simpleFunc();
```

In this example, the `simpleFunc()` function was called without an argument, so the value of the parameter `param1` in the function was initialized with the default value `undefined`. However, such an example should not set a precedent. The responsibility here is on you to decide what should happen if fewer or no arguments have been used. For example, you can respond to that in the following way:

```
function simpleFunc(param1) {
    if (param1 === undefined) {  // Has an argument been passed?
        console.log("simpleFunc(): No argument received!")
    } else {
        console.log(param1);
    }
}
```

The question as to how you should respond when a function with multiple parameters is called with fewer arguments, depends, of course, on the function itself. If multiple values are expected for a mathematical function, you'll probably have to issue a warning or error message.

As an alternative, you can continue using a function if you use default values for omitted arguments. JavaScript knows the principle of *default parameters*, where the default value is used if no corresponding argument was passed to the function. Here's a simple example you can use to create a user from first name and last name. Thanks to the default parameters you can call this function with none, one, or two arguments:

```
function userTemplate(fname = "John", lname = "Doe") {
    let user = {
        userfname: fname,
        userlname: lname
    }
    return user;
}

let user1 = userTemplate();
console.log(user1.userfname);  // Output: John
console.log(user1.username);  // Output: Doe
let user2 = userTemplate("Jason");
console.log(user2.userfname);  // Output: Jason
console.log(user2.username);  // Output: Doe
let user3 = userTemplate("Jason", "Wolfe");
console.log(user3.userfname);  // Output: Jason
console.log(user3.userlname);  // Output: Wolfe
```

Listing 18.3 /examples/chapter018/18_1_2/script.js

When you call a function with arguments, you can access them in the function via the `arguments` object. The `arguments` object is an array-like object with `arguments[n]`, where `n` represents the number of arguments. The first argument starts with `arguments[0]`. You can determine the number of arguments passed to the function using `arguments.length`. The following example demonstrates the use of `arguments`:

```
function sumAll() {
    let sum = 0;
    if (arguments.length === 0) {  // Have no arguments been passed?
        return 0;  // ... then end function with 0
    }
    for (let i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
let sum = sumAll(100, 200, 123, 300, 55);
console.log("Result=" + sum);  // Output: Result=778
```

Listing 18.4 /examples/chapter018/18_1_2/script2.js

In this example, all values passed to the `sumAll()` function are added using the `arguments` object. Before that, a check is run to see if any arguments have been passed at all. However, in practice, such tasks are now more likely to be implemented using the *rest parameter*.

The rest parameter also allows you to use a function with any number of function parameters. The rest parameter is a real array. The following function demonstrates the rest parameter:

```
function sumAll(iVal, ...myargs) {
    let sum = iVal;
    myargs.forEach(function(val) {
        sum += val;
    });
    return sum;
}
console.log(sumAll(100, 200, 300, 400));  // Output: 1000
```

Listing 18.5 /examples/chapter018/18_1_2/script3.js

The first parameter in this example—`iVal`—is still an ordinary parameter with the value `100`. The remaining parameters are defined using `...myargs` and stand for the rest (here, the values `200`, `300`, and `400`). We implement the access to the remaining parameters of the `myargs` array in the example via the `forEach()` method, which calls the function written in it for each value of the `myargs` array. In each case, the current value is passed to `val` as a parameter to the function written in it. In the example, each value in the `myargs` array is added to the `sum` variable and returned at the end.

This function can be made somewhat clearer if we use the arrow function notation:

```
let sumAll = (iVal, ...myargs) => {
    let sum = iVal;
    myargs.forEach((val) => sum += val);
    return sum;
}
```

The syntax of the arrow function notation will be explained in more detail in Section 18.1.5.

18.1.3 Return Value of a Function

If you want to return a value from a function, you must use a `return` statement. You can use the `return` statement to specify the value to be returned. Functions without a `return` statement use a default value, which is again `undefined` in most cases. After a `return` statement in a function, the execution returns to the caller, where often the return value of a variable is assigned. You've already seen and used multiple examples of this.

It's not an error to use a function without parentheses when calling it because a function can be passed or referenced to a variable in this way. For example:

```
let isDebug = true;

function debugMessage() {
    if (isDebug) {
        return 'Debug mode is active';
    }
    return 'Debug mode disabled';
}

let msg = debugMessage; // Assign function to variable
console.log(typeof msg); // function
let txt = msg();         // call debug_message()
console.log(txt);        // Output: Debug mode is active
isDebug = false;
console.log(msg());      // Output: Debug mode disabled
```

Listing 18.6 /examples/chapter018/18_1_3/script.js

In this example, the `debug_message` function was passed to the `msg` variable, rather than calling the `debug_message` function and passing the return value to `msg`, as might have been expected. In this assignment, you've created a `msg` variable that references the `debug_message` function object and, like `debug_message()`, can be called explicitly as a function by using `msg()`.

18.1.4 The Scope of Variables in a Function

At this point, I need to mention an indispensable topic, namely the scope or visibility of variables. At the same time, I'll also solve the mystery about `let` and `var` before variables. Each function creates a new scope, but at first that's not a block scope, as is the case in other programming languages. The following example will demonstrate what that means:

```
let iVal = 111;  // Global vriable

function simple(param1) {
    if (param1) {
        var sVal = 222;
    }
    return sVal + iVal;  // Okay, thanks to variable hoisting
}

let sumUp = simple(true);
console.log(sumUp);
console.log(sVal + iVal);  // Error: sVal unknown
```

The `iVal` variable is a global variable and visible everywhere in the entire script. Consequently, this variable can be used both within a function and outside of it. I don't think that's a big surprise to anyone. The `sVal` variable, on the other hand, is only visible inside the `simple()` function, which is why using it outside of the function, as is done at the end of the example with `console.log(sVal + iVal)`, leads to the error message that `sVal` is unknown. This is the scope JavaScript creates for each function. However, JavaScript doesn't create a block scope within the `if` block here, as is usually the case in other programming languages. JavaScript uses variable hoisting and interprets the example as follows:

```
function simple(param1) {
    var sVal;
    if (param1) {
        sVal = 222;
    }
    return sVal + iVal;
}
```

Therefore, the `sVal` variable is visible within the entire function. This behavior—that a variable is declared via `var` inside a statement block within a function and is also visible outside the block inside the function—isn't always desirable and can entail errors. In addition, it often makes it harder to understand the code. For this reason, the better alternative is the `let` keyword, which should be used instead of `var`. A variable created with `let` gets a block scope and is thus only visible in the current code block. In our example, you can implement a block scope as follows:

```
...
function simple(param1) {
    if (param1) {
        let sVal = 222; // Block scope with let
        console.log(sVal); // sVal now only valid inside the if block
    }
    return sVal + iVal; // Error: sVal now also unknown here
}
…
```

The `let` keyword enables you to restrict the scope of a variable to individual code blocks, as is the case by default in many other programming languages.

If you had used neither `let` nor `var` before the `sVal` variable in this example, you'd have defined a global variable you could have accessed from anywhere, even from outside the function. Usually you don't want this at all, and it can be avoided even with strict mode by always writing a `"use strict";` at the beginning of the JavaScript code.

Strict Mode for Functions

It's also possible to switch only one function and not the entire JavaScript to strict mode. To do that, you only need to specify the corresponding `"use strict";` (or `'use strict';`) statement within the function as the first statement. For example:

```
function simple() {
  "use strict";
  // Code for the simple() function
}
```

In JavaScript, you can also define functions within functions so that they're only visible and valid within the function. The following example calls a function to divide two values. Inside the function, `normalize()` is called, which checks if one of the values is 0, and in that case makes it a 1. The reason is that a division by 0 makes no sense. Calling the `normalize()` function outside the `divide()` function would result in an error because it isn't visible there. Here's the example:

```
function divide(x, y) {
    return normalize(x) / normalize(y);

    function normalize(val) {
        if (val == 0) {
            return 1;
        }
        return val;
    }
}
console.log(divide(4, 0));
```

Listing 18.7 /examples/chapter018/18_1_4/script.js

18.1.5 Defining Functions in Short Notation (Arrow Functions)

A more modern way to define functions with relatively little effort is available with arrow functions. In addition to the more compact notation, the main advantage of arrow functions is that the `this` keyword within the function refers to the context in which the function was defined, and not, as in a normal function, to the context in which the function is executed. For this purpose, here's a simple example first:

```
let double = val => val * 2;
console.log(double(100)); // Output: 200
```

If you use multiple parameters with the arrow function notation, you must put the parameters in parentheses:

```
let sum = (param1, param2) => param1 + param2;
console.log(sum(100, 300)); // Output: 400
```

You already know the shorter version of the function from the introduction:

```
let sum = function(param1, param2) {
    return param1 + param2;
}
```

Surely, you've noticed that the function body has been omitted in the arrow functions. This is possible as long as there's only one statement in the function. If you use multiple statements, you must also use curly brackets here, for example:

```
let debug = msg => {
    console.log("Debug output -> ");
    console.log(msg);
    console.log("<- Debug output")
}

let val = 1234;
debug("Current value val (" + val + ")");
```

The same applies to the `return` statement, which you can omit in a short form of the arrow function without a function body. The following principle applies here: once the function consists of multiple lines, you must use a `return` statement, for example:

```
let multiplication = (param1, param2) => {
    console.log("Multiplication is being executed");
    return param1 * param2;
}
console.log(multiplication(10, 5));
```

The only thing missing is the syntax for a function without parameters, where you have to use the empty function parentheses `()`. For example:

```
let simple = () => console.log("Function without parameters");
simple();
```

18.1.6 Using a Function in a Web Page

To make sure that this chapter doesn't get too theoretical, we'll demonstrate how you can call a function within a web page. A few things are anticipated here, but I figure this simple example is still comprehensible. In the example, you can enter two values each via HTML form elements and either add them together via a button or multiply them.

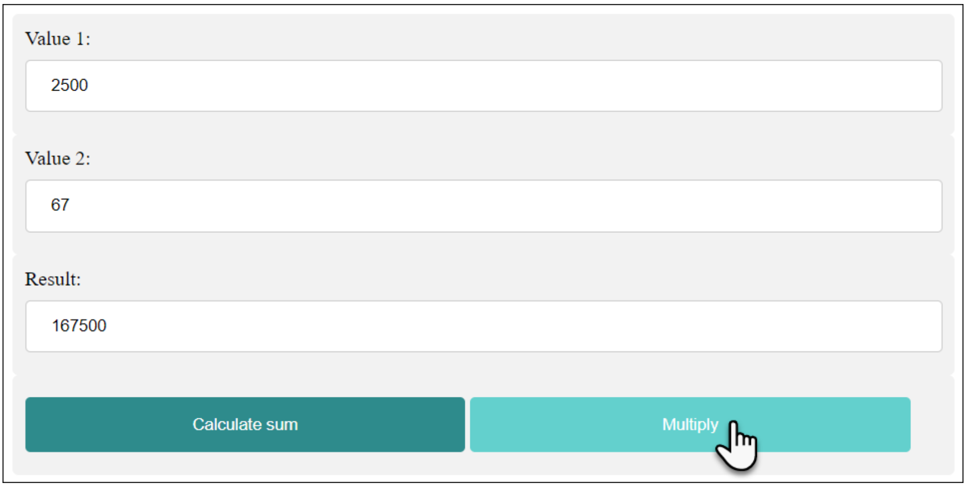
An event listener (`document.addEventListener()`) intercepts which button was clicked (`click event`), and then a corresponding JavaScript function gets called—`calculateSum()` for an addition and `calculateMul()` for a multiplication. In the function itself, the corresponding values of the `input` element are read, calculated, and passed to the JavaScript function `showResult()` for output of the calculation to a separate text field. The JavaScript code for this is shown in Listing 18.8, and part of the HTML code is printed in Listing 18.9. You can see the website in use in Figure 18.1.

```
document.addEventListener('DOMContentLoaded', function() {
    let button1 = document.getElementById('button-calculate-sum');
    button1.addEventListener('click', calculateSum);
    let button2 = document.getElementById('button-calculate-mul');
    button2.addEventListener('click', calculateMul);
});

function calculateSum() {
    let x = parseInt(document.getElementById('field1').value);
    let y = parseInt(document.getElementById('field2').value);
    let result = x + y;
    showResult(result);
}

function calculateMul() {
    let x = parseInt(document.getElementById('field1').value);
    let y = parseInt(document.getElementById('field2').value);
    let result = x * y;
    showResult(result);
}

function showResult(result) {
    let resultField = document.getElementById('result');
    resultField.value = result;
    console.log(result);
}
```

Listing 18.8 /examples/chapter018/18_1_6/js/calc.js

```
...
<body>
    <div>
        <label for="field1">Value 1: </label>
        <input id="field1" type="text" value="5">
    </div>
    <div>
        <label for="field2">Value 2:</label>
      <input id="field2" type="text" value="5">
    </div>
    <div>
        <label for="result">Result: </label>
        <input id="result" type="text">
    </div>
    <div>
        <button id="button-calculate-sum">Calculate sum</button>
        <button id="button-calculate-mul">Multiply</button>
    </div>
    <script src="js/calc.js"></script>
</body>
```

Listing 18.9 /examples/chapter018/18_1_6/index.html



Value 1:

2500

Value 2:

67

Result:

167500

Figure 18.1 JavaScript Functions When Executed within a Web Page

18.2 Arrays

If you want to store multiple values in one variable, you can do this with an array. You create an array by assigning comma-separated values to a variable in square brackets. This notation is also referred to as *array literal notation*. Let's look at a simple example:

```
let user = ["John", "Frank", "Steven"];
```

This way, you can create an array with the identifier `user` and assign it three strings with user names. Surely, in this case, you could also use individual variables instead, as follows:

```
let user01 = "John";
var user02 = "Frank";
var user03 = "Steven";
```

In practice, using single variables is more complex and cumbersome than using an array. For example, what happens if you want to loop through multiple names to select a single name? And what do you do when you need 100 user names instead of 3? Here, you're much better off with an array because you can manage many values with only one identifier.

If there are many entries in the array, it's useful to write each entry in one line. This makes the array clearer. In practice, therefore, the following notation would be recommended:

```
let user = [
   "John",
   "Frank",
   "Steven",
   "Peter",
   "Jay"
];
```

Unlike other programming languages, arrays in JavaScript can also contain entries with different data types. Thanks to loose typing, an array is also allowed as follows:

```
let user = [
    "Wolfe",
    46,
    "email@email.com",
    false
];
```

You can create an empty array as follows:

```
let user = [];  // An empty array
```

Besides the array literal notation, you can also create an array using a constructor function via `new`:

```
let user = new Array();  // An empty array
```

You can also create an array with a specific size right away. For example, you create an array with 10 elements as follows:

```
let user = new Array(10); // 10 undefined elements
```

However, specifying the size of an array isn't really necessary with arrays because an array can grow dynamically at runtime. Watch out when you create an array as follows:

```
let vals = new Array(10, 50); // 2 elements vals[0]:10, vals[1]:50
```

In that case, you create an array with two occupied elements. You can specify the length of the array only if you call the constructor function of `Array()` with a single decimal value as an argument.

Likewise, in the constructor function notation, you can already pass values directly as arguments:

```
let user = new Array("John", "Frank", "Steven");
```

For the arrays with `Array()`, you can also omit the keyword `new`. In practice, you'll have to deal a lot with arrays in the future. Whether you prefer to use the array literal notation or the constructor function notation is up to you. I personally prefer the shorter array literal notation with the square brackets.

18.2.1 Accessing the Individual Elements in the Array

You can access the individual elements of an array using the square brackets and the corresponding index number. The first element in an array always has index [0], the second element index [1], and so on. For example:

```
let user = [
    "John",   // [0]
    "Frank",  // [1]
    "Steven"  // [2]
];
console.log(user[1]); // Output: Frank
let name01 = user[0]; // name01 = "John"
console.log(name01); // Output: John
user[2] = "Steve"; // "Steve" gets overwritten
console.log("user[0] = " + user[0]); // Output: user[0] = John
console.log("user[1] = " + user[1]); // Output: user[1] = Frank
console.log("user[2] = " + user[2]); // Output: user[2] = Steve
```

Listing 18.10 /examples/chapter018/18_2_1/script.js

Here's a simple example of how to get the current day of the week by name using an array with all the days of the week, the `Date` object, and the `getDay()` method:

```
let date = new Date();
let day = date.getDay();
let wd = [
    "Sunday",     // wd[0]
    "Monday",     // wd[1]
    "Tuesday",    // wd[2]
    "Wednesday",  // wd[3]
    "Thursday",   // wd[4]
    "Friday",     // wd[5]
    "Saturday"    // wd[6]
];
console.log("Today is " + wd[day]);
```

Listing 18.11 /examples/chapter018/18_2_1/script2.js

You've already learned about the `Date` object and the `getDay()` method. In this example, you've added a `wd` array with all seven days of the week as individual strings. The order from Sunday with the value 0 to Saturday with the value 6 was kept just as the method `getDay()` returns a corresponding value from 0 to 6 of the day. If the example is executed on a Wednesday, the `day` variable is assigned the value 3 from `date.getDay()`, so using `wd[day]` in this case is equivalent to `wd[3]` and therefore uses the string `"Wednesday"` from the array.

18.2.2 Multidimensional Arrays

You can also use an array within another array, which is very useful, for example, if you need to manage data records that are always the same. You already know this principle from a spreadsheet. For example, the following data is given for a user administration:

```
let user = [
    "pronix74",  // Nickname
    46,  // Age
    "wolfe@pronix.com", // Email
    false  // Administrator rights
];
```

Object or Array?

Such structures are usually not represented as arrays, but as objects.

Here, you have an array for a user, including nickname, age, email, and whether this user has admin rights. If you now want to create multiple users with the same data, you can do this using a multidimensional array as follows:

```
let user = [
    ["pronix74", // [0][0]
        46, // [0][1]
        "wolfe@pronix.com", // [0][2]
        false // [0][3]
    ],
    ["halwa66", // [1][0]
        51, // [1][1]
        "halwa@pronix.com", // [1][2]
        false // [1][3]
    ],
    ["woafu", // [2][0]
        46, // [2][1]
        "1@woafu.com", // [2][2]
        true // [2][3]
    ]
];
```

Now you've created a multidimensional array with three users. The access works with the usual one-dimensional array over the index. With the first dimension, you specify the user as index (here, 0, 1, or 2), and with the second dimension, you access the desired value. For example, to access the data of the second user in the `user` array, you can do the following:

```
...
console.log(user[1][0]); // Output: halwa66
console.log(user[1][1]); // Output: 51
console.log(user[1][2]); // Output: halwa@pronix.com
console.log(user[1][3] ? "Admin" : "User"); // Output: User
```

Listing 18.12 /examples/chapter018/18_2/script.js

18.2.3 Adding or Removing New Elements in an Array

For adding and removing elements in an array, there are useful methods that make your life easier. Of course, it's also possible to operate quite conventionally with the index operator on an array, but this also creates the risk that you produce undefined holes in the array. Here's an example of the Spartan way:

```
let user = [
    "John",  // [0]
    "Jason", // [1]
    "Ben"    // [2]
];

user[3] = "Tom";
user[5] = "Jay";
user[2] = undefined; // Delete value

for (let i = 0; i < user.length; i++) {
    console.log(user[i]);
}
```

Listing 18.13 /examples/chapter018/18_2_3/script.js

In the example, holes were created for the elements `user[2]` and `user[4]`. The content of these elements is `undefined`. Of course, you can check in a loop where empty space exists (= `undefined`) and then insert the element there, but it's much easier to keep the array without holes right away. In the example, the `for` loop pass also uses the `length` property, which contains the number of elements in an array.

Traversing Arrays Conveniently Using "for ... in" and "for ... of"

Instead of fiddling with the `length` property, however, you can use the `for ... in` or `for ... of` loop to run through the individual elements of an array. In the following example, you can see the `for ... in` loop in use, which outputs all elements from the `user` array:

```
let user = [
    "John",  // [0]
    "Jason", // [1]
    "Ben"    // [2]
];

for (let n in user) {
    console.log(user[n]);
}
```

The `for ... of` loop is a little more comfortable. It allows you to run only through the property values of the iterable properties and to omit a cumbersome `user[n]`. Here's the same example again, but now with `for ... of`:

```
let user = [
    "John",  // [0]
    "Jason", // [1]
    "Ben"    // [2]
];

for (let n of user) {
    console.log(n);
}
```

A Brief Overview of Common Methods for Adding and Removing Elements in the Array

For adding and removing elements, JavaScript provides specific methods. Table 18.1 contains a list of some common methods that are often used for this purpose.

Method	Description
`pop()`	Removes the last element in the array
`push()`	Inserts a new element at the end of the array
`shift()`	Removes the first element of an array
`unshift()`	Inserts an element at the beginning of the array
`slice()`	Removes elements from an array
`splice()`	Adds, replaces, or deletes element(s) at any position in the array

Table 18.1 Common Methods of Arrays

Adding and Removing Elements at the End: “push()” and “pop()”

To add elements at the end of the array, you can call the `push()` method, while the `pop()` method enables you to remove the last element. The argument you need to pass to the `push()` method is the element that is to be added, or you can pass several elements at once. The `push()` method returns the length of the array. Its counterpart `pop()`, on the other hand, doesn’t need an argument and always removes the last element in the array. As a return value, `pop()` returns the removed element or `undefined` if there’s no element left in the array. Here’s a simple example to demonstrate `push()` and `pop()` in use:

```
let user = []; // Empty array

user.push("pronix74");
user.push("halwa66");
console.log(user.length); // Output: 2
let size = user.push("root01", "scotty33", "anonymus");
```

```
console.log(size) // Output: 5

for (let n of user) {
    console.log(n);
} // Output:  pronix74, halwa66, root01, scotty33, anonymus

let n = user.pop(); // Remove last element -> anonymus
console.log(n + " was removed")
user.pop(); // Remove last element again
console.log(user.length); // Output: 3

for (let n of user) {
    console.log(n);
} // Output:  pronix74, halwa66, root01
```

Listing 18.14 /examples/chapter018/18_2_3/script2.js

Using an Array as a Stack

In programming, the `push()` and `pop()` methods are often used to use an array as a stack according to the last in, first out (LIFO) principle, which means that the last element placed on the stack with `push()`is always removed first with `pop()`. This can be used, for example, to implement the **Undo** function, which can be used to undo the last user action.

Adding and Remove Elements at the Beginning

The counterparts of `push()` and `pop()` are `unshift()` and `shift()` for adding and removing elements at the beginning of the array. The `unshift()` method works basically the same as `push()`, except that the element or elements are added at the beginning. This method also returns the new length of the array. The counterpart `shift()`, on the other hand, removes the first element in the array and returns it as a return value. Here's a simple example to show these two methods in practice:

```
let user = []; // Empty array

user.unshift("pronix74");
user.unshift("halwa66");
console.log(user.length); // Output: 2
let size = user.unshift("root01", "scotty33", "anonymus");
console.log(size) // Output: 5

for (let n of user) {
    console.log(n);
} // Output:  root01, scotty33, anonymus, halwa66, pronix74

let n = user.shift(); // Remove first element -> root01
console.log(n + " was removed")
user.shift(); // Remove first element again
console.log(user.length); // Output: 3

for (let n of user) {
    console.log(n);
} // Output:  anonymus, halwa66, pronix74
```

Listing 18.15 /examples/chapter018/18_2_3/script3.js

Using an Array as a Queue

A queue is a data structure based on the first in, first out (FIFO) principle, where the first element in the queue is always returned, and new elements are added at the end. It's the classic principle of a queue at the checkout. In practice, this data structure can be implemented by using the `push()` method to add the new elements at the end and the `shift()` method to return and remove the first element from the array. Of course, you can also write a queue the other way around, using `pop()` and `unshift()` instead of `push()` and `shift()`. A classical method for applying a queue are message queues, that is, queues for messages that are transmitted from the sender to a receiver.

Inserting and Removing Elements at Any Position in the Array: “splice()”

To add, remove, or replace elements from an array, you can use the `splice()` methods. The method allows for multiple arguments to be passed to it. The first argument is the relevant position where the new element should be added, removed, or replaced. As a second argument, you can specify the number of elements to be deleted from this position. If you want to only insert elements, you can use `0`. The remaining arguments represent the elements you want to insert or replace in the array. The following example demonstrates all three options of `splice()` in use:

```
let user = [
    "pronix74",
    "halwa66",
    "root01"
];
user.splice(2, 0, "anonymus"); // Insert at user[2].

for (let n of user) {
    console.log(n);
} // Output: pronix74, halwa66, anonymus, root01
```

```
let del = user.splice(1, 2); // delete [1]&[2]
console.log(del + " were deleted!");
user.splice(1, 0, "woafu86", "john123"); // insert 2 elements
user.splice(0, 1, "pronix1974") // replace user[0] with pronix1974

for (let n of user) {
    console.log(n);
} // Output: pronix74, woafu86, john123, root01
```

Listing 18.16 /examples/chapter018/18_2_3/script4.js

18.2.4 Sorting Arrays

The `sort()` method allows you to sort arrays. The advantage of this method is that you can use it to write your own function that sets the sorting criteria. You can define the comparison function with two parameters that are called internally in pairs for the values of the array when `sort()` is called. With an appropriate return of -1, 1, or 0, the `sort()` method then takes care of sorting the array. Return -1 if the value is greater than the second value. The opposite is true if you return 1. With a return value of 0, both values are equal.

The comparison function that sorts two strings (also works with numeric values) by their size (alphabet) looks as follows:

```
function compare(val1, val2) {
    if (val1 < val2) {
        return -1; // val is less than val2
    } else if (val1 > val2) {
        return 1; // val1 is greater than val2
    } else {
        return 0; // val1 and val2 are identical
    }
}
```

"localCompare()"

I recommend that you use the `localCompare()` function for strings. This is part of the ECMAScript standard and compares the strings depending on the country-specific settings of the system.

A special feature in JavaScript is that you can pass this comparison function to the `sort()` method as an argument. This isn't possible in all programming languages. I'll get back to that momentarily. You can sort an array of strings alphabetically using the `sort()` method and the `compare()` function as follows:

```
...
let user = [
    "halwa66",
    "pronix74",
    "conrad22",
    "anton43",
    "beta88"
];
user.sort(compare);
for (let n of user) {
    console.log(n);
} // Output: anton43, beta88, conrad22, halwa66, pronix74
```

Because functions are also objects in JavaScript, they can be used as arguments or even return values just like variables, as you've seen with the `sort(compare)` function. With regard to the previous examples, you can also use this to implement the loop pass with the output of the individual elements as a function. For this, too, the arrays in JavaScript provide a useful and simple option: the `forEach()` method. The passed argument with this method gets output with each element. You can use it to output each element of an array via a function with `forEach()`:

```
...
function printUser(item) {
    console.log(item);
}
let user = [
    "halwa66",
    "pronix74",
    "conrad22",
    "anton43",
    "beta88"
];
user.sort(compare);
user.forEach(printUser);
```

Listing 18.17 /examples/chapter018/18_2_4/script.js

18.2.5 Searching within Arrays

Before you write a function to search for a specific element in the array, let me introduce you to two functions for that as well. There's nothing wrong with doing the following:

```
...
for (let n in user) {
    if (user[n] === "anton43") {
        console.log("Found at position: " + n);
    }
}
...
```

But for this purpose, JavaScript provides `indexOf()`, which enables you to search directly in the array for a specific element. You can specify the element to search for as an argument. Alternatively, you can use a second argument that determines from which index the search should be started. If -1 is returned, then the element isn't contained in the array. For example:

```
console.log(user.indexOf("anton43"));
let pos = user.indexOf("hilary");
if (pos === -1) {
    console.log("Could not find hilary!")
}
```

The `indexOf()` method starts searching at the beginning of the array. But if you want to start the search at the end of the array, you can use the `lastIndexOf()` method instead. The method is used in the same way as `indexOf()`. Other suitable methods such as `find()` and `findIndex()` should also be mentioned here. The `find()` method either returns the value of the element of an array that fulfills the condition of a provided test function, or it returns `undefined`. The `findIndex()` method either returns the index of the first element in the array that satisfies the provided test function, or it returns –1.

18.2.6 Additional Methods for Arrays

With the methods presented here, you should be very well equipped to get started with JavaScript for now. However, there are a lot of other methods available to you. I've listed some other useful methods in Table 18.2.

Method	Description
`concat()`	Allows you to append elements or arrays to another array.
`copyWithin()`	Allows you to copy elements within the array.
`find()`	Allows you to search for elements according to search criteria. The search criteria are passed as an argument. The element which is found gets returned.

Table 18.2 Other Useful Methods to Work with Arrays

Method	Description
`findIndex()`	Like `find()`, except that it returns the index of the first occurrence.
`filter()`	Allows you to sort out elements from the array according to certain filter criteria. Here, too, you pass the filter criteria in the form of a function as an argument.
`join()`	Converts an array into a string.
`reverse()`	Sorts the elements in the array into the reverse order.
`slice()`	Allows you to cut out individual elements from an array.

`toString()`
`toLocaleString()`
`valueOf()`
	Enable you to convert arrays into strings.

Table 18.2 Other Useful Methods to Work with Arrays (Cont.)

18.3 Strings and Regular Expressions

You've already gotten to know the strings in the data types. If you're a web developer using JavaScript, strings are usually the most common type of data you'll use. For this reason, I'll devote a few sections to them here.

The internal structure of strings isn't unlike that of arrays. The first character of a string also starts with the index 0. The string "Hello world" is internally constructed as shown in Figure 18.2.

H	E	L	L	O		W	O	R	L	D
0	1	2	3	4	5	6	7	8	9	10

Figure 18.2 The Internal Structure of a String

18.3.1 Useful Functions for Strings

You'll often want to check the length of an input string in an input `field` of a web form. For this purpose, the string data type provides the `length` property, which contains exactly the number of characters in a string. Here's an example:

```
let hello = "Hello world";
console.log(hello.length); // Output: 11
```

For searching, you can also use the `indexOf()` and `lastIndexOf()` methods for the string data type. The first occurrence of a searched character or string will be returned. As with the array, the `indexOf()` method starts at the beginning of the string, while `lastIndexOf()` starts at the end of the string. Let's look at a simple example:

```
let hello = "Hello world";
console.log(hello.indexOf("lo")); // Output: 3
console.log(hello.indexOf(" ")); // Output: 5
console.log(hello.indexOf("World")); // Output: -1
```

The `substring()` and `substr()` methods are available for extracting strings. Both methods expect the start index from where the string should be extracted as the first argument. If you don't specify a second argument, the extraction will be performed from the start index to the end. If you do specify a second argument, then for `substring()`, it should be the index up to which to extract. With `substr()`, on the other hand, the second argument must be the number of characters to be extracted from the starting index. Here's another simple example:

```
let hello = "Hello world";
let pos1 = hello.indexOf(" ");
console.log(hello.substring(pos1 + 1)); // Output: world
console.log(hello.substr(3, 2)); // Output: lo
console.log(hello.substring(0, pos1)); // Output: Hello
```

Other useful methods are `toLowerCase()` and `toUpperCase()` to convert all characters of a string to lowercase and uppercase, respectively.

18.3.2 Applying Regular Expressions to Strings

Regular expressions are used to describe a pattern of strings to formulate a search expression. A regular expression is a formal language that can be used to describe a (sub)set of strings and that can be used, for example, in search and/or replace operations.

Regular expressions are objects of the `RegExp` type and can be created either as a constructor function with `new RegExp()` or as a literal notation. Let's look at a simple example:

```
let txt = "This text is being searched";
let regEx01 = /Text/; // Literal notation
let regEx02 = new RegExp(/will/); // Constructor function
let n01 = txt.search(regEx01); // Search for "text" in txt
console.log('"text" found at pos. ' + n01);
let n02 = txt.search(regEx02); // Search for "will" in txt
console.log('"will" found at pos. ' + n02);
let newText = txt.replace(regEx01, "paragraph"); // Search and replace
console.log(newText); // Output: The search takes place in this paragraph.
```

For regular expressions in JavaScript, there are of course many more options available besides the properties and methods shown here.

18.4 Object-Oriented Programming in JavaScript

To better understand the principle of objects in JavaScript, it's probably best to start by learning how to create your own objects. Once you understand the underlying concept, you won't have any more problems with the predefined objects or the browser objects. Of course, it should be noted here that this is only an introduction to object-oriented programming (OOP) in JavaScript.

18.4.1 What Exactly Are Objects?

Simply put, objects in JavaScript are basically nothing more than complex and compound variables with properties and methods. The properties of an object are also called *attributes* or *properties* and the methods are sometimes also referred to as *object methods*. The object provides you with access to all properties and methods. Based on what we've learned so far about JavaScript in this book, we can also say that properties are the data and methods are the functions of an object. Take a look at the following JavaScript code as an example:

```
function print(usr) {
    console.log("Nickname    : " + usr[0]);
    console.log("Age         : " + usr[1]);
    let admin = usr[2] ? "Yes" : "No";
    console.log("Admin rights : " + admin);
}

let user = [
    "pronix74", // Nickname
    46, // Age
    false // Admin rights
];
print(user);
```

In this example, the variables and the function for the output were used separately. The disadvantage of such somewhat unstructured scripts is that they use a loose collection of global variables and functions. Such scripts are difficult to adapt or extend later. As the amount of data and functions grows, it will become confusing over time.

With very little effort, you can create or encapsulate a template for an object with properties and methods from these global variables and functions, and you only need one global object through which the rest of the data and functions can then be accessed.

The following adapted example is intended to show you how easy it is to apply OOP in JavaScript:

```
let user = {
    nickname: "pronix74", // Nickname
    age: 46, // Age
    admin: false, // admin rights
    print: function() { // "function()" is optional
        console.log("Nickname : " + this.nickname);
        console.log("Age     : " + this.age);
        console.log("Admin    : " + this.isAdmin());
    },
    isAdmin: function() {
        return this.admin ? "Yes" : "No";
    }
};
user.print();
```

Listing 18.18 /examples/chapter018/18_4_1/script.js

In this example, you can see one of several ways to create an object with JavaScript using the *object literal notation*. Here, you can see that objects in JavaScript are implemented using key-value pairs. Key and value are separated by a colon. You can use the key to access the corresponding values. A value itself can in turn be a literal, a function, or some other object. The key is the identifier for the property or method of the object.

Everything after the assignment to `let user` between the curly brackets is the content of the object. In the example, there are three data properties, that is, `nickname`, `alter`, and `admin`, and two methods, `print` and `isAdmin`. In JavaScript, methods are also introduced with the keyword `function`. Since ES6, however, the keyword can also be omitted. You must separate the individual properties and methods of an object with commas, and the value of a property or method must be written after a colon.

The Keyword "this"

I'll describe the keyword `this` separately later on. Within the methods `print()` and `isAdmin()`, it represents the object on which the method is executed. Without `this`, access within the methods to the properties wouldn't work.

18.4.2 Creating Objects via Constructor Functions

Alternatively, you can define a constructor function and create a new object using the `new` keyword. Unlike the object literal notation, the constructor function allows you to create any number of copies of the object. In JavaScript itself, there's no constructor for creating instances, as are available in other OOP languages, which is why a function is used as a constructor function here. Basically, the appearance of a constructor function doesn't differ from a normal function. To distinguish it from a normal function, it's recommended to capitalize the first letter in constructor functions.

Here's a constructor function from which you can subsequently create any number of objects:

```
function User(nickname, age, admin) {
    this.nickname = nickname;
    this.alter = age;
    this.admin = admin;
    this.printUser = function() {
        console.log("Nickname : " + this.nickname);
        console.log("Age     : " + this.age);
        console.log("Admin    : " + this.isAdmin());
    }
    this.isAdmin = function() {
        return this.admin ? "Yes" : "No";
    }
};

let user01 = new User("pronix74", 46, false);
let user02 = new User("halwa66", 52, true);
user01.printUser();
user02.printUser();
```

Listing 18.19 /examples/chapter018/18_4_2/script.js

To turn a function into a constructor function, you must first call it using the keyword `new`. The function then creates a new object and returns it. You don't need to use `return` for a constructor function because the new object gets returned implicitly. The properties and methods within the constructor function are accessed via the keyword `this`.

18.4.3 Creating Objects via the Class Syntax

Because objects in particular take some getting used to for those switching from another object-oriented language to JavaScript, ECMAScript 6 (2015) introduced a class syntax that's quite similar to that of Java or C++, for example. If you come from these programming languages, you've implemented such object types using classes. JavaScript doesn't know classes, but uses constructor functions and prototypes for this purpose.

For this reason, the `class` keyword was introduced to define the "class", while the `function` keyword (optional since ES6 anyway) is no longer needed for the methods.

Here's the `User` example in a class syntax:

```
class User {
    constructor(nickname, age, admin) {
        this.nickname = nickname;
        this.alter = age;
        this.admin = admin;
    }
    print() {
        console.log("Nickname : " + this.nickname);
        console.log("Age     : " + this.age);
        console.log("Admin    : " + this.isAdmin());
    }
    isAdmin() {
        return this.admin ? "Yes" : "No";
    }
};

let user01 = new User("pronix74", 46, false);
let user02 = new User("halwa66", 52, true);
user01.print();
user02.print();
```

Listing 18.20 /examples/chapter018/18_4_3/script.js

The `constructor()` method is called implicitly when you create a new object instance of the corresponding class. Basically, the `constructor()` method corresponds to the constructor function from the previous section. Again, no `return` is required because the object instance is returned implicitly. Apart from that, creating a new object works the same way as with the constructor function with `new`, which implicitly calls the `constructor()` method of the class.

18.4.4 Accessing the Object Properties and Methods: Setters and Getters

To access object properties and methods, we usually use the dot notation, as you've already seen in the examples before. Consider this example:

```
...
let user01 = new User("pronix74", 46, false);
// Dot notation:
user01.print();
console.log(user01.nickname); // Output: pronix74
```

Besides the dot notation, you can also write the property and method in square brackets between single or double quotes. Here's the alternative option:

```
...
let user01 = new User("pronix74", 46, false);
// Bracket notation:
user01["print"]();
console.log(user01["nickname"]); // Output: pronix74
```

You can use the dot notation or the bracket notation as you like. The only thing I'd recommend is that you keep it consistent. I personally prefer the dot notation. Nevertheless, there are individual cases in which you must use the bracket notation. This happens when properties or methods contain a hyphen, for example. Without the square bracket notation an error would occur because the minus sign would be interpreted as a subtraction operator.

In this way, you could now also change the properties of an object directly. If you want to change the nickname, for example, this could be done as follows:

```
user01.nickname = "woafu1974";
user01.age = "I won't tell";
user01.admin = "root";
```

You can already see from the example that it would now also be possible to enter any nonsense. In OOP, setter methods are usually used to change individual properties, while getter methods are used to retrieve individual properties. Especially with the setter methods, you can then still check the passed value for its validity.

JavaScript provides the keywords `set` (for setter methods) and `get` (for getter methods). You must place the keyword before the method. For example, to change the nickname, you must use `set`. To retrieve individual properties, on the other hand, you would use `get`. However, you aren't forced to implement both versions. Here's the example with the setter and getter methods in use:

```
class User {
    constructor(nickname, age, admin) {
        this._nickname = nickname;
        this._age = age;
        this._admin = admin;
    }

    isAdmin() {
        return this._admin ? "Yes" : "No";
    }
    set nickname(name) {
        if (typeof name === "string") {
            this._nickname = name;
        } else {
            console.log("Error: no string!")
        }
    }
    get nickname() {
        return this._nickname;
    }
    set age(age) {
        if (typeof age === "number") {
            this._age = age;
        } else {
            console.log("Error: Not an integer!")
        }
    }
    get age() {
        return this._age;
    }
    set admin(adm) {
        if (typeof adm === "boolean") {
            this._admin = adm;
        } else {
            console.log("Error: true oder false!")
        }
    }
    get admin() {
        return this._admin;
    }
};

let user01 = new User("pronix74", 46, false);
// Setter methods in use
user01.nickname = "woafu1974"; // -> set nickname("woafu1974")
user01.age = 47; // -> set age(47)
user01.admin = true; // -> set admin(true)
// Getter methods in use
console.log(user01.nickname); // -> get nickname()
console.log(user01.age); // -> get age()
console.log(user01.admin); // -> get admin();
```

Listing 18.21 /examples/chapter018/18_4/script.js

To avoid naming conflicts between the properties and methods here, it's common practice to have the properties begin with an underscore. Thanks to this data encapsulation, you now also have the properties better protected against access from outside, although it's still possible to address the properties directly from outside with an underscore (e.g., `user01._nickname`).

18.4.5 The Keyword "this"

You've already used the keyword `this` quite a bit, so I'll briefly go into some more detail here. In simple terms, `this` is a kind of property which, when called, is assigned the value of the object with which it's called. In other words, it's virtually a reference to itself. `this` is therefore an implicit parameter that is automatically available.

`this` is a keyword whose value is resolved and usually references an object. However, the actual value of `this` again depends on the execution context in which it was called.

In the following example, `this` is written in a class, a global function, and the global environment. Within the class, `this` refers to the instantiated `SimpleClass` object itself. For the global function, the execution context is `undefined` in strict mode. Without strict mode, it would be the `window` object when the JavaScript is executed in the web browser. When using `this` globally, on the other hand, the global `window` object is used by default, but only if the program is run in a web browser.

```
...
class SimpleClass {
    simple() {
        console.log(this);  // Output: SimpleClass
    }
};

function simpleFunction() {
    console.log(this);  // Output: Window (in browser)
                        // Output: undefined (with Node.js)
}

let val1 = new SimpleClass();
val1.simple();
simpleFunction();
console.log(this); // Output in web browser: window
```

Listing 18.22 /examples/chapter018/18_4_5/script.js

`this` is thus a reference to the owner of the object, which is necessary because there can be and are more instances of an object. Especially within methods of a created object, this is very important because only in this way can you make sure that with several instances of the same objects also the stored properties of the individual instance can be accessed. After all, each individual instance of an object has its own properties and doesn't share them with any other instance.

18.5 Other Global Objects

Predefined objects are the native objects that JavaScript provides as part of the language. These ready-made objects provided by JavaScript include `Array`, `Boolean`, `Date`, `Function`, `Map`, `Set`, `Math`, `Number`, `RegExp`, `String`, and `Object`. You already got to know some of them in the previous sections. In the following sections, I'll briefly describe the objects of JavaScript that haven't been mentioned yet.

18.5.1 The Top Object "Object"

This is the object from which all other objects in JavaScript are derived. Every object in JavaScript is of the `Object` type in any case and may belong to more specific types (e.g., `String`) as well. The `Object` object provides properties and methods that are available for all other objects to work with. Basically, an object of the `Object` type is just a container for data such as strings or numbers, but you can also place function objects into it. You can create an object in the long notation using `new Object` as follows:

```
let data = new Object();
data.name = "John Doe";
data.account number = 34234123;
data.aba = 7200032123;
```

Or you can just write it as usual as an object literal:

```
let data = {
  name: "John Doe",
  Account number: 34234123,
  ABA: 7200032123
};
```

18.5.2 Objects for the Primitive Data Types: Number, String, and Boolean

You've already used the primitive data types for numbers, strings, and truth values regularly in this book. Example:

```
let iVal = 1234;
console.log(typeof iVal);   // Output: number
let str  = "string";
console.log(typeof str);    // Output: string
let bool = false;
console.log(typeof bool);   // Output: Boolean
```

In JavaScript, there are also full-fledged object versions for the primitive data types, for which appropriate methods are provided for the type. Here's an example with the object versions for the primitive data types:

```
let iOVal = new Number(1234);
console.log(typeof iOVal);  // Output: object
let oStr  = new String("String");
console.log(typeof oStr);   // Output: object
let oBool = new Boolean(false);
console.log(typeof oBool);  // Output: object
```

In practice, however, it's recommended not to use the object version of the primitive data types, but to leave it at the primitive form. The object version only makes the code more complicated and slows down the execution speed of the script. Things get more complicated when you compare a variable with the primitive data type and an object version. For example:

```
let iVal = 1234;
let iOVal = new Number(1234);

if ( iVal == iOVal) {
  console.log("The value of iVal is equal to iOVal.");
}
else {
  console.log("The value of iVal is not equal to iOVal.");
}

if ( iVal === iOVal ) {
  console.log("The value and type of iVal is equal to iOVal.");
}
else {
  console.log("The value and type of iVal is not equal to iOVal.");
}
```

Listing 18.23 /examples/chapter018/18_5_2/script.js

When comparing with the == operator, `true` gets returned because JavaScript performs an automatic conversion here. A comparison with the === operator, on the other hand, returns `false` because the value is the same, but the type is different. In practice, it's recommended to use the === operator in JavaScript for a direct comparison because, in most cases, you don't want to have automatic type conversion. Usually, you want to have a strict comparison of two values, taking type and value into account.

If you want to use the methods of `String`, `Number`, or `Boolean` objects that are provided, you don't need to create a `String`, `Number`, or `Boolean` object manually via `String()`, `Number()`, or `Boolean()` because as soon as you call a method with the primitive data types, the primitive data type will get converted into a corresponding object. For example:

```
let str  = "string";
console.log("Number of characters: " + str.length); // 12
console.log(str.toUpperCase());                    // CHARACTER STRING

let iVal = 1234;
console.log("1234 as dual number        : " + iVal.toString(2));  // 10011010010
console.log("1234 as hexadecimal number : " + iVal.toString(16)); // 4d2
console.log("1234 as octal number       : " + iVal.toString(8));  // 2322
```

Due to the automatic type conversion, JavaScript converts the primitive data types into a transient object here. The `str` variable temporarily becomes a `String` object, and `iVal` temporarily becomes a `Number` object when the corresponding methods for `String` and `Number` objects are called, respectively.

18.5.3 "Function" Object

I've already mentioned that functions in JavaScript are also objects. All functions in JavaScript are based on `Function`. By using the `Function` object, you can access useful properties and some methods. In practice, you could theoretically also create a function using the constructor function via `new Function()`, but this form is relatively rarely used and needed.

18.5.4 "Date" Object

The `Date` object provides you with an extensive set of methods for different calculations of date and time. To use the `Date` object, you need to create a new object via the constructor function, `new Date()`. Internally, JavaScript uses the milliseconds that have elapsed since January 1, 1970, at 00:00:00 as the unit. Let's take a look at a simple example:

```
let date = new Date();
console.log(date);              // 2023-02-25T17:01:17.952Z
console.log(Date.parse(date)); // ms since 1/1/1970; 00:00
```

Note that the system time refers to the time at the user whose web browser or general runtime environment is running the script. The current system time of the server computer isn't determined here.

18.5.5 "Math" Object

For various types of calculations, the `Math` object is available with useful features and methods. In practice, you don't need to create a `Math` object via the constructor function `new Math()`, but you can use the features and methods directly via `Math.Property` or `Math.Method()`. Take a look at this example.

```
console.log("Constant for Pi : " + Math.PI);  // 3.141592653589793
let r = 12;
let a = r * r * Math.PI; // calculate circular area
console.log(a);
console.log(Math.random()); // generate pseudo-random number between 0 and 1
```

18.5.6 "Map" Object

`Map` allows you to create ordered lists from key-value pairs. Such a `map` is often referred to as an associative array. In this context, the keys and the values can consist of any data type. There are many useful methods available to use the entries. The following example demonstrates a `map` using zip codes:

```
let zip = new Map([
    [97217, "Portland"],
    [60647, "Chicago"],
    [02114, "Boston"],
    [77007, "Houston"]
]);

let zipTmp = zip.get(97217);
console.log(zipTmp); // Output: Portland
// Add key-value pair
zip.set(94112, "San Francisco");
```

Listing 18.24 /examples/chapter018/18_5_6/script.js

18.5.7 "Set" Object

A `Set` object is a collection of values of any type that are stored in the order in which they are added. The highlight is that each value is unique in a `Set`. An algorithm internally checks for equality before adding. You can add individual elements using `add()`. If you use `has()`, you can check if an element already exists in the set. You can determine the number of elements via `size()`. For this purpose, here's a simple example with the different methods related to `Set`:

```
let mySet = new Set([1, 3, 5]);
mySet.add(7); // Added at the end
mySet.add(3); // Will not be added, already exists
mySet.add("some text"); // Add to the end

// Check
console.log(mySet.has(5)); // = true
console.log(mySet.has(9)); // = false
console.log(mySet.size); // = 5
```

```
// Iterate through a Set and output
for (let item of mySet) console.log(item);

// Delete element
mySet.delete(3);
mySet.delete("some text");
```

Listing 18.25 /examples/chapter018/18_5_7/script.js

18.6 Summary

The focus of this chapter was completely on functions, arrays, and objects in JavaScript. You now know how you can write your own routines by using functions and how to manage multiple values at the same time by using arrays. As for OOP in JavaScript, you've learned what objects are and how to create your own objects with properties and methods in JavaScript. In addition to the objects you've created yourself, you also have become familiar with the predefined objects provided by JavaScript as a language.

Chapter 19

Changing Web Pages Dynamically

Once a web page has been loaded, the web browser generates the Document Object Model (DOM) from the page. This enables you to dynamically generate HTML using JavaScript, which is a good reason to look into the topic of DOM and DOM manipulation.

The *Document Object Model* (*DOM*) allows you to access all HTML elements of the document via JavaScript. This, in turn, enables you to use JavaScript to manipulate all HTML elements, the HTML attributes, and even all CSS styles of a web page. Additionally, you can add new HTML elements or attributes and remove existing ones. Likewise, the DOM makes it possible to respond to all existing HTML events of a web page.

In a nutshell, here's what you'll learn in this chapter:

- How to search for specific HTML elements
- How to change the content of an HTML element
- How to change the stylesheet of HTML elements
- How to respond to events of the DOM
- How to add new HTML elements to the document
- How to change or even remove existing elements
- How to access the individual values of a form element with JavaScript

19.1 Introduction to the DOM of an HTML Document

You already know from HTML that the HTML elements of a document are composed into a hierarchical tree structure. At the top of the tree, you'll find the `document` object, followed by the root of the tree, which is usually the `html` element. Below the `html` element, you'll find the `head` and `body` elements.

In Figure 19.1, you can see the diagram of such a hierarchical tree structure. There you can see in a family tree how HTML documents are logically represented. All individual elements of this family tree are *nodes* and are related to each other. For example, the `head` element has two children: the `title` and `meta` elements.

Because the `title` and `meta` elements are the descendants of the same parent element, they are also referred to as *siblings*. The same is true for the descendants of the `body` element, where in the figure, the `header`, `nav`, `article`, and `footer` elements have the same parent element (`body`) and are thus also siblings.

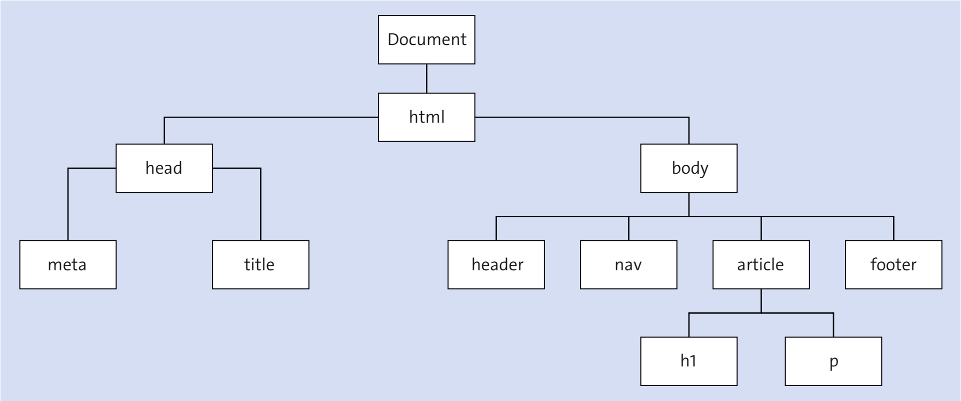
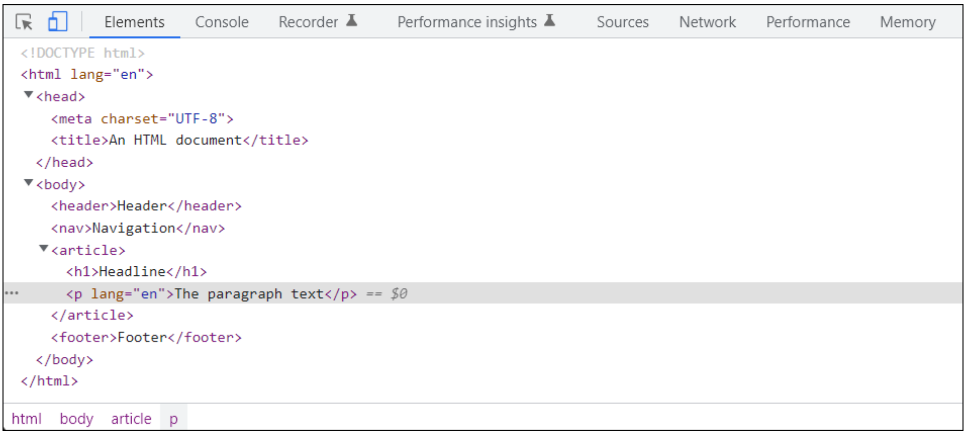

Document
html
head
body
meta
title
header
nav
article
footer
h1
p

Figure 19.1 Diagram of a DOM Tree with Objects

It's not only the HTML elements that represent a node in a DOM tree. The HTML attributes and the contents of the HTML elements themselves are also nodes of a DOM tree that can be accessed using JavaScript. The following simple HTML construct shows all three important node types:

```
<p lang="en">The paragraph text ...</p>
```


Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory

```
<!DOCTYPE html>
<html lang="en">
▼<head>
   <meta charset="UTF-8">
   <title>An HTML document</title>
 </head>
▼<body>
   <header>Header</header>
   <nav>Navigation</nav>
  ▼<article>
     <h1>Headline</h1>
     <p lang="en">The paragraph text</p> == $0
   </article>
   <footer>Footer</footer>
 </body>
</html>
```

html
body
article
p

Figure 19.2 The Representation of the Tree Structure in the DOM Inspector of the Web Browser

Here, you have the HTML element node with the `p` element. Additionally included is the HTML attribute node with `lang="en"`. In addition, the content (here, `The paragraph text ...`) of the `p` element is a genuine node (also called a *text node*) that can be accessed in the DOM tree using JavaScript.

Due to this division into node objects, where all HTML elements, HTML attributes, and the content represent a node, and these nodes are related to each other in the tree by parent, child or sibling relationships, it's possible to access each of these nodes using various DOM methods and DOM properties. On the following pages, you'll learn how this works.

19.2 The "document" Object

As you can see in Figure 19.1, the `document` object is the topmost object of the DOM tree. This `document` object enables you to access all elements of the HTML document with JavaScript and change them under certain circumstances. In this context, the `document` object represents the complete web page and is the owner of all other nodes of the web page. If you want to address an element in an HTML document, for example, you can do that via the `document` object and the `querySelector()` method as follows:

```
let element = document.querySelector('body');
```

If you need access to other objects (nodes) of the web page, you need to start at the top with the `document` object.

19.3 DOM Programming Interface

As you've already learned, you can use JavaScript to access the individual nodes of the DOM. For this, DOM provides various methods and properties for each object in the DOM tree. Here's a short example that demonstrates the interaction of the programming interface of the DOM with a method and a property. First the HTML code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DOM interfaces</title>
</head>
<body>
  <h1>The DOM interface</h1>
  <p>The paragraph text</p>

  <script src="js/script.js"></script>
</body>
</html>
```

Listing 19.1 /examples/chapter019/19_3/index.html

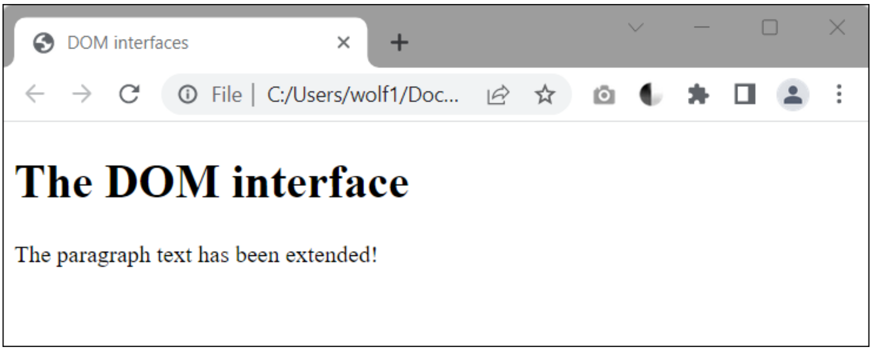
And here's the JavaScript code:

```
let text = document.querySelector('p').innerHTML;
if (text) {
    text += " " + "has been extended!";
    document.querySelector('p').innerHTML = text;
}
```

Listing 19.2 /examples/chapter019/19_3/js/script.js

In this example, `querySelector()` is a method and `innerHTML` is a property of the `document` object. The `querySelector()` method is used to get access to an HTML element. The `innerHTML` property, on the other hand, can be used to read the content of the HTML element or replace it with new content.

In this example, you search for the first `p` element in the current HTML document using the `querySelector()` method and assign the content contained in the `innerHTML` property to the `text` variable. After this assignment, `text` contains the string, `"The paragraph text"`. In the second line, you expand the contents of `text` so that the complete string is `"The paragraph text has been extended!"`. In the last line of the script, you change the content (text node) of the `p` element and pass or replace the old content with the newly extended content using the `innerHTML` property with the `text` string. You can see the example during execution in Figure 19.3.



The DOM interface

The paragraph text has been extended!

Figure 19.3 The Content of a <p> Element Was Manipulated Using the "querySelector()" Method and the "innerHTML" Property

19.4 Accessing Elements in the DOM

JavaScript is commonly used to read, modify, or extend elements in the DOM. For those types of access to the DOM and its element nodes, the `document` object provides several methods.

Method	Description
`document.getElementById()`	Finds an element based on the `id` attribute
`document.getElementsByTagName()`	Finds all elements with a specific tag name
`document.getElementsByClassName()`	Finds all elements of a given CSS `class`
`document.getElementsByName()`	Finds HTML elements with a specific “name” attribute
`document.querySelector(s)`	Returns the first element (and only this one) that corresponds to the specified CSS selector `s`
`document.querySelectorAll(s)`	Returns a list with all elements that match the specified CSS selector `s`

Table 19.1 Methods for Finding HTML Elements

Although there are several methods available to you to access individual elements with JavaScript, in practice, the two still somewhat newer methods `querySelector()` and `querySelectorAll()` are usually sufficient. They are also much easier to use because you can use them to search for the usual CSS selectors such as elements, classes, and IDs, as well as other attributes.

19.4.1 Finding an HTML Element with a Specific “id” Attribute

The old way to access a node in the `document` object is to search for a specific `id` attribute of an element using the `getElementById(id)` method. This method returns a reference to the element object if successful, or it returns `null` if no element with this `id` attribute exists. Here’s the same example again, but this time with an `if` check that queries whether the element was found.

First, here’s the important HTML part:

```
...
<h1>The DOM interface</h1>
<p id="msg">The paragraph text</p>

<script src="js/script.js"></script>
...
```

Listing 19.3 /examples/chapter019/19_4_1/index.html

Then, here’s the JavaScript code:

```
let elem = document.getElementById('msg');
if (elem) {
    let text = elem.innerHTML;
    text += " " + "has been extended!";
    elem.innerHTML = text;
} else {
    console.log("Element with ID msg was not found!");
}
```

Listing 19.4 /examples/chapter019/19_4_1/index.html

In this example, `if()` is first used to check the condition as to whether a corresponding ID (here, `msg`) is contained in the HTML document at all. If that's true, the element will be manipulated. If it isn't true, the `else` branch will be executed, and a corresponding error message will be output to the console.

As briefly indicated at the beginning, this method of accessing elements with the IDs and JavaScript is somewhat outdated. In the past, one simply used as many `id` attributes as possible, which then, together with the class names for the presentation, made for a rather extensive and confusing HTML document. You can avoid inflating the HTML document with IDs and classes for JavaScript by simply using the appropriate (semantic) elements.

In addition, accessing the element via `getElementById()` is quite cumbersome and could be done right away using the `querySelector` method as the better alternative:

```
...
  let elem = document.querySelector('#msg');
...
```

Listing 19.5 /examples/chapter019/19_4_1/js/script-2.js

The use of `querySelector('#msg')` shown here corresponds to that with `getElementById('msg')`. In addition, `querySelector()` also makes it much clearer to access the common CSS selectors, such as `#msg`, for an ID here.

19.4.2 Finding HTML Elements with a Specific Tag Name

If you're looking for HTML elements with a specific tag name, you can do this by using the `getElementsByTagName()` method. From the method name in the plural (`getElements`), you can probably already guess that not only will one element in the HTML document be returned but also a collection of all nodes with a corresponding tag name. You can access the individual nodes via square brackets and the corresponding index value. The number of elements found will be returned by the `length` property.

"getElementsByTagName()" Doesn't Return an Array

Although it may seem so, `getElementsByTagName()` doesn't return an array, but a *node list* (*live NodeList*), which you can read with a loop. You can't call array-type methods such as `forEach()` directly on a node list.

Here's a simple example that demonstrates how you can use `getElementsByTagName()`. First of all, here's the HTML document again:

```
...
<article id="lead">
  <h2>The DOM interface</h2>
  <p>First paragraph text in the article</p>
  <p>Second paragraph text in the article</p>
</article>

<p>First paragraph text outside the article</p>
<p>Second paragraph text outside the article</p>

    <h2>Output:</h2>
    <output></output>
    <script src="js/script.js"></script>

...
```

Listing 19.6 /examples/chapter019/19_4_2/index.html

And here's the JavaScript code for it:

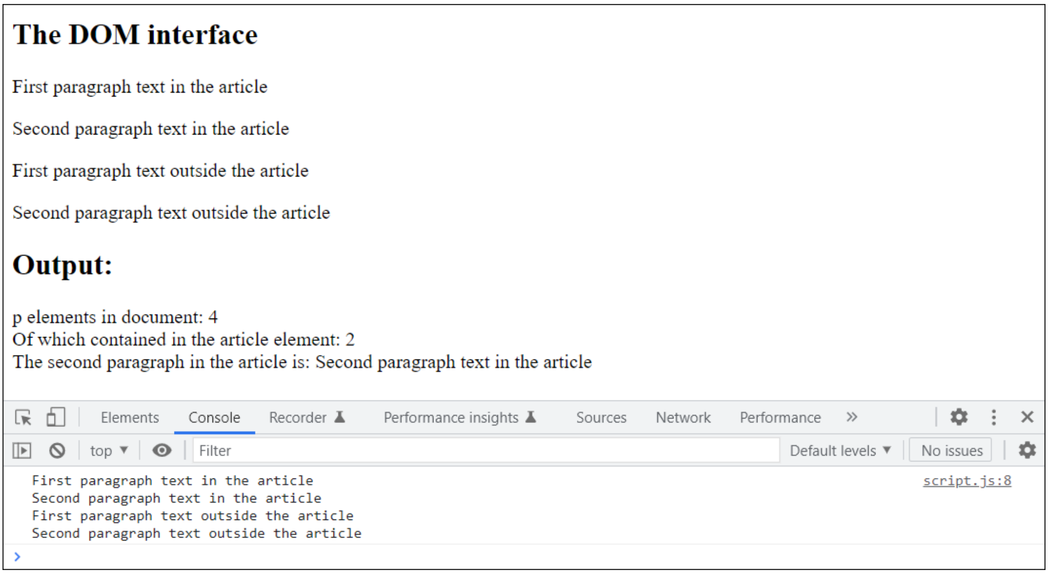
```
let plainText = "";
let pElements = document.getElementsByTagName('p');
for (let i = 0; i < pElements.length; i++) {
    plainText += pElements[i].innerHTML + '\n';
}
console.log(plainText); // Output for demonstration

let htmlText = "p elements in document: " + pElements.length + "<br>";
let articleElements = document.getElementById('lead');
let articlePElements;
if (articleElements) {
    articlePElements = articleElements.getElementsByTagName('p');
    htmlText += "Of which contained in the article element: " +
                articlePElements.length + "<br>";
}
```

```
htmlText += "The second paragraph in the article is: " +
            articlePElements[1].innerHTML;
document.querySelector('output').innerHTML = htmlText;
```

Listing 19.7 /examples/chapter019/19_4_2/js/script.js

At the beginning, after calling `let pElements = document.getElementsByTagName('p');`, all found `p` elements of the HTML document are contained in `pElements`. `pElements.length` contains the number of `p` elements in a `for` loop, and the `length` property as a termination condition allows you to use `console.log()` to output the individual elements found with the index in the square brackets and the `innerHTML` property to the JavaScript console for demonstration. Then, we still incorporate this information into the `htmlText` string with the number of `p` elements found in the HTML document.

You use `let articleElements = document.getElementById('lead');` to search for an element where the attribute value of `id` is equal to `lead`. The returned node gets saved in `articleElements`. Using the node, you can search with `articlePElements = articleElements.getElementsByTagName('p');` for all `p` elements that are inside the node. In the example, the attribute value `id='lead'` is used for the `article` element. Thus, in `articlePElements`, you'll find all `p` elements that are contained within the `article` element. You also append this information to the `htmlText` string. Last but not least, we'll demonstrate how you can access the individual contents directly using the index (here, with `articlePElements[1].innerHTML`). This text was also added to the end of the `htmlText` string. You can see the result of the example during execution in Figure 19.4.



The DOM interface

First paragraph text in the article

Second paragraph text in the article

First paragraph text outside the article

Second paragraph text outside the article

Output:

p elements in document: 4
Of which contained in the article element: 2
The second paragraph in the article is: Second paragraph text in the article

```
First paragraph text in the article
Second paragraph text in the article
First paragraph text outside the article
Second paragraph text outside the article
```

Figure 19.4 Demonstrates the "getElementsByTagName()" Method, Which Returns All Nodes of a Certain Tag Name (Here, "p")

If you look at the following JavaScript lines in the example, you'll probably agree that this is a relatively awkward solution after all:

```
...
let articleElements = document.getElementById('lead');
let articlePElements;
if (articleElements) {
    articlePElements = articleElements.getElementsByTagName('p');
    htmlText += "Of which contained in the article element: " +
                articlePElements.length + "<br>";
}
...
```

Listing 19.8 /examples/chapter019/19_4_2/js/script.js

First, `getElementById()` is used to search for an element with the ID `lead` and, in case of a find within `if()`, `getElementsByTagName()` is used to search for the individual `p` elements. Of course, you could just give the `p` elements a class name and then search for them, but again, `querySelectorAll()` provides a method that makes searching for the right elements a breeze. `querySelectorAll()` works like `querySelector()`, but this method returns a list of found elements. `querySelector()`, on the other hand, returns only the first element found that matches the CSS selector. Thus, the preceding lines can be simplified as follows by using `querySelectorAll()`:

```
...
let articlePElements = document.querySelectorAll('#lead p');
if (articlePElements) {
    htmlText += "Of which contained in the article element: " +
                articlePElements.length + "<br>";
}
...
```

Listing 19.9 /examples/chapter019/19_4_2/js/script-2.js

This has the same effect as the preceding example and returns all `p` elements that are within the ID `#lead`.

19.4.3 Finding HTML Elements with a Specific "class" Attribute

If you're looking for an HTML element with a specific CSS class name assigned with the HTML attribute `class`, you can use the `getElementsByClassName()` method to do so:

```
let myc = document.getElementsByClassName('aClass');
```

Like `getElementsByTagName()`, this method returns all found nodes in the HTML document with the class name `aClass`. Because `getElementsByClassName()` works just like `getElementsByTagName()`, except that it searches for nodes with a specific `class` attribute value, we don't need an extra example here.

Here, too, it's now convenient to use the much more universal `querySelectorAll()` method in the following way to find all CSS classes:

```
let myc = document.querySelectorAll('.aClass');
```

19.4.4 Finding HTML Elements with a Specific "name" Attribute

The `getElementsByName()` method is the version to search for nodes in the HTML document that contain the HTML attribute `name` with a specific value. Again, all found nodes are stored in a list, which you can access again with the corresponding index in square brackets. The `name` attribute is mainly used in form elements and can be used, for example, to evaluate related radio buttons. Take a look at the following example. First, the HTML document:

```
...
<input name="color" type="radio" value="Red">Red
<input name="color" type="radio" value="Blue">Blue
<input type="button" onclick="getColor()" value="Choose color"><br>
<output></output>

<script src="js/script.js"></script>
...
```

Listing 19.10 /examples/chapter019/19_4_4/index.html

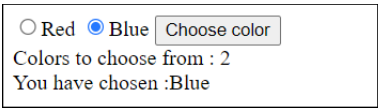
And here's the JavaScript for it:

```
function getColor() {
    let colors = document.getElementsByName('color');
    let htmlText = "Colors to choose from : " + colors.length +
        "<br>You have chosen :";
    if (colors[0].checked) {
        htmlText += "Red";
    } else if (colors[1].checked) {
        htmlText += "Blue";
    } else {
        htmlText += "None";
    }
    document.querySelector('output').innerHTML = htmlText;
}
```

Listing 19.11 /examples/chapter019/19_4_4/js/script.js

In the example, the event handler `onclick` is used as an HTML attribute in the HTML tag and will execute the event function `getColor()` when the element (here, the button) is clicked. I'll describe event handlers separately, but you can already see that they're an important link between HTML and JavaScript.

The statement `let colors=document.getElementsByName("color");` in `getColor()` makes sure that all nodes are found where `name="color"` is written and stored in `colors`. You can use the `if` conditions (`colors[i].checked`) to check whether the corresponding radio button has been activated (= `true`) or not activated (= `false`). You'll learn more about HTML forms with JavaScript later in this chapter. The example during execution is shown in Figure 19.5.


○ Red ● Blue

Colors to choose from : 2
You have chosen :Blue

Figure 19.5 Evaluation of Radio Buttons Using the "getElementsByName()" Method

You'll certainly ask yourself now whether that's also possible with `querySelectorAll()`. And indeed, instead of `getElementsByName()`, you can use the `querySelectorAll()` method here as well, as follows:

```
...
let colors = document.querySelectorAll('[name="color"]');
...
```

Listing 19.12 /examples/chapter019/19_4_4/js/script-2.js

19.4.5 Using "querySelector()" and "querySelectorAll()"

If you've read everything from the beginning up to this point, you should already have noticed that, in practice, you really only need `querySelector()` and `querySelectorAll()` to select elements based on CSS selectors. Not only are these two methods more flexible than the `getElementById()` and `getElementsByTagName()` methods, they're also faster. The `querySelector()` method returns the first element found, while `querySelectorAll()` returns a list of all elements found in a `NodeList`.

Because you can select the elements using CSS selectors with `querySelector()` and `querySelectorAll()`, even more complex accesses can be implemented relatively easily. Let's take a look at the following example:

```
...
  <article>
    <h2>Article Heading 1</h2>
    <p>The 1st paragraph text</p>
  </article>
  <article>
    <h2>Article Heading 2</h2>
    <p>The 2nd paragraph text</p>
  </article>
    <article>
    <h2>Article Heading 3</h2>
    <p>The 3rd paragraph text</p>
  </article>
  <article>
    <h2>Article Heading 4</h2>
    <p>The 4th paragraph text</p>
  </article>

  <h2>Heading 5 (no article)</h2>
  <p>The 5th paragraph text</p>

  <script src="js/script.js"></script>
...
```

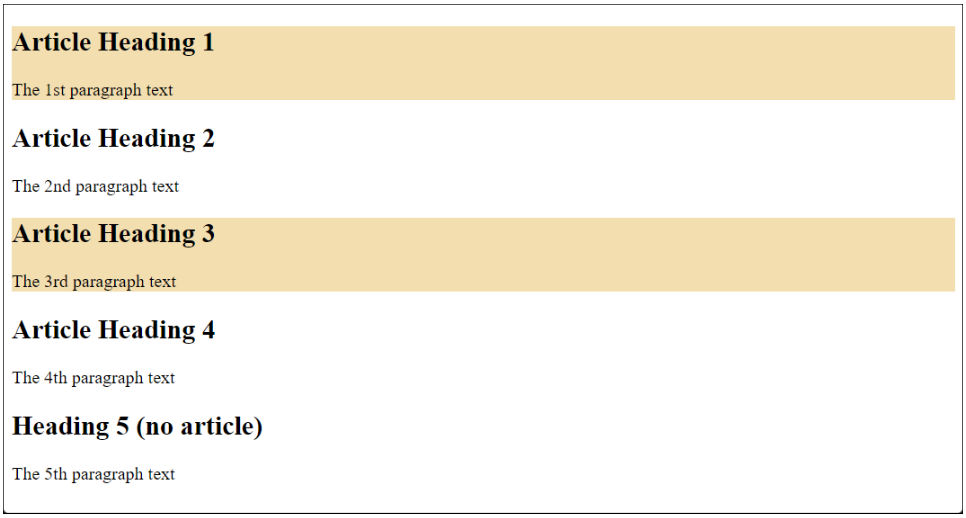
Listing 19.13 /examples/chapter019/19_4_5/index.html

If, in this example, you want to alternate the `article` elements with special background color, this is easier than you might think if you use `querySelectorAll()`. Here's the corresponding example:

```
let elem = document.querySelectorAll('article:nth-child(odd)');
for (let i = 0; i < elem.length; i++) {
  elem[i].style.backgroundColor = "wheat";
}
...
```

Listing 19.14 /examples/chapter019/19_4_5/js/script.js

As you can see in Figure 19.6, each `article` element has been styled with a wheat-colored background color if it's an odd element (`nth-child(odd)`) of the parent element. Thus, in this example, the first, third, fifth, seventh, and so on element would be styled. On the other hand, if you want to additionally style all even elements, you just need to use `article:nth-child(even)` instead. The fifth heading with the paragraph text wasn't styled because it isn't an `article` element. Here, you could also see how easy it is to change the style of an HTML element via DOM. To learn more about changing the style of an HTML element via DOM, Section 19.5.3.



Article Heading 1

The 1st paragraph text

Article Heading 2

The 2nd paragraph text

Article Heading 3

The 3rd paragraph text

Article Heading 4

The 4th paragraph text

Heading 5 (no article)

The 5th paragraph text

Figure 19.6 The “querySelector()” and “querySelectorAll()” Methods Provide a Flexible Way to Access DOM Elements

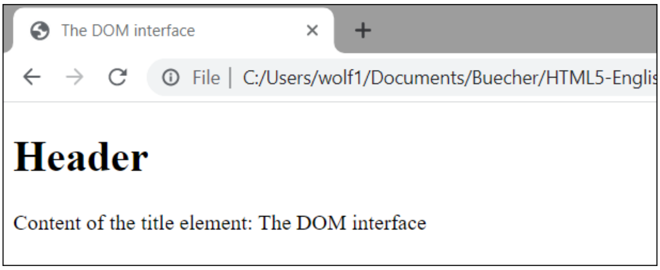
19.4.6 Other Object and Property Collections

In addition, ready-made object collections and properties are available to help you find HTML elements. An overview of this is shown in Table 1.2. But in a small example, you’ll first see how you can use these ready-made collections:

```
let evaluation = document.querySelector('output');
if (evaluation) {
    evaluation.innerHTML = "Content of the title element: " + document.title;
}
```

Listing 19.15 /examples/chapter019/19_4_6/js/script.js

The example returns the content of the `title` element with `document.title`, as you can see in Figure 19.7.



Header

Content of the title element: The DOM interface

Figure 19.7 You Can Use “document.title” to Determine the Content of the <title> Element

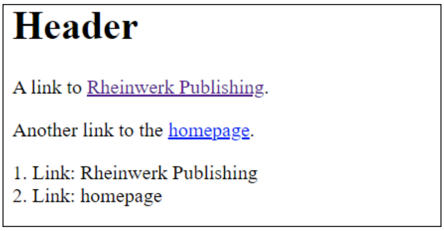
Of course, the object and property collection is only a convenience over the other ways of locating an HTML element described earlier. To access the content of the `title` element, you could just as well have used on of the following two options instead of `document.title`:

```
document.querySelector('title').innerHTML
document.querySelector('title').textContent
```

"innerHTML" versus "textContent"

The `textContent` property contains only the text content of an HTML element and all child elements. With the `innerHTML` property, on the other hand, the HTML code is used in addition to the text content.

Other object collections, in turn, return an entire list of values. The following example returns all links of the HTML document.



Header

A link to Rheinwerk Publishing.

Another link to the homepage.

1. Link: Rheinwerk Publishing
2. Link: homepage

Figure 19.8 Finding All Hypertext Links in an HTML Document

The HTML code for this follows:

```
...
<p>A link to
   <a href="https://www.rheinwerk-verlag.de/">Rheinwerk Publishing</a>
</p>
<p>Another link to the
  <a href="/html/standard/index.html">homepage</a>
</p>

<output></output>
<script src="js/script-2.js"></script>
...
```

Listing 19.16 /examples/chapter019/19_4_6/index-2.html

Here's the corresponding JavaScript:

```
let hyperlinks = document.links;
let text = "";
```

```
for (let i = 0; i < hyperlinks.length; i++) {
    text += i + 1 + ". Link: " + hyperlinks[i].innerHTML + "<br>";
}
document.querySelector('output').innerHTML = text;
```

Listing 19.17 /examples/chapter019/19_4_6/js/script-2.js

Table 19.2 contains an overview of all object and property collections that can make your life with DOM easier with regard to searching for HTML elements.

Method	Description
`document.baseURI`	Returns the absolute base URI of the HTML document.
`document.body`	Returns the `body` element.
`document.cookie`	Returns all cookies of the document.
`document.doctype`	Returns the `doctype` of the document.
`document.documentElement`	Returns the `html` element.
`document.documentURI`	Returns the URI of the document.
`document.domain`	Returns the domain name from the document server.
`document.domConfig`	Returns the DOM configuration.
`document.embeds`	Returns a list of all `embed` elements.
`document.forms`	Returns a collection of all `form` elements.
`document.head`	Returns the `head` element.
`document.images`	Returns a collection with all images.
`document.implementation`	Returns the DOM implementation.
`document.inputEncoding`	Returns the character set (encoding) of the document.
`document.lastModfied`	Returns the date and time when the document was last modified.
`document.links`	Returns a collection of all links with the `a` and `area` element that contain a value in the `href` attribute.
`document.readyState`	Returns the load status of the document.
`document.referrer`	Returns the URI of the linking document. This assumes that the document to be linked has been accessed via a link. If an address was selected directly or via a bookmark, `document.referrer` is empty.

Table 19.2 Overview of Ready-Made Object and Property Collections

Method	Description
`document.scripts`	Returns a list with all `script` elements.
`document.title`	Returns the `title` element.
`document.URL`	Returns the complete URL of the document.

Table 19.2 Overview of Ready-Made Object and Property Collections (Cont.)

19.5 Changing an HTML Element, an Attribute, or the Style

Several properties are available to change the content of an HTML element. Table 19.3 contains a brief overview of these properties or methods, which will be described separately later in this chapter.

Method or Property	Description
`element.`**`innerHTML`**`=`	Changes the content of an HTML element
`element.`**`attribute`**`=`	Changes the value of an HTML attribute
`element.`**`setAttribute(attr,val)`**	Changes the value of an HTML attribute
`element.`**`style.`***`property`*`=`	Changes the style (CSS) of an HTML element

Table 19.3 Various Properties and a Method for Changing HTML Elements

19.5.1 Changing the Content of HTML Elements Using “innerHTML”

You can change the content of HTML elements using the `innerHTML` property, which is something you’ve already done numerous times in this book. All HTML elements, except those without opening and closing HTML tags (e.g., `<img>`), have this `innerHTML` property. In the following example, the content of an `h1` and a `p` element is again manipulated using `innerHTML`:

```
...
<h1>Heading</h1>
<p>Paragraph text</p>

<button onclick="changeContent()">
  Change with innerHTML
</button>

<script scr="js/script.js"></script>

...
```

Listing 19.18 /examples/chapter019/19_5_1/index.html

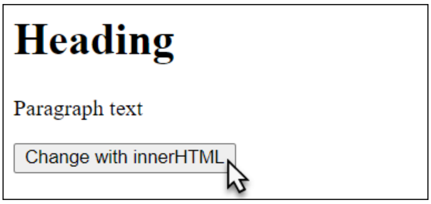
Here’s the JavaScript for it:

```
function changeContent() {
    document.querySelector('h1').innerHTML = "New Heading!";
    let elem = document.querySelector('p');
    elem.innerHTML = "New content for the paragraph text";
}
```

Listing 19.19 /examples/chapter019/19_5_1/js/script.js

In this HTML document, you’ll find an `h1` element and a `p` element. You can search for both elements using the `querySelector()` method. Then, you can change the content of these elements using `innerHTML`. Here, I’ll also present two different ways to access the content of HTML elements: one directly without and one indirectly by using a variable.

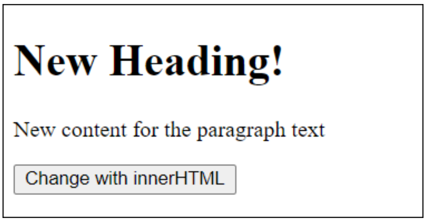
Figure 19.9 shows the example in its original state, and Figure 19.10 shows the example after changing the content of the `h1` and `p` elements with `innerHTML` by clicking the button and calling the JavaScript function `changeContent()` you wrote yourself.



Heading

Paragraph text

Figure 19.9 The HTML Document in Its Original State



New Heading!

New content for the paragraph text

Figure 19.10 The Example after Changing the Content of the <h1> and <p> Elements with “innerHTML”

Setting HTML Tags in “innerHTML”

Besides the option to simply put a text with `innerHTML` into an HTML element, you can also use HTML tags. The trick is that these tags will also be rendered as elements in the HTML document. For example, with regard to the example shown here, if you want to highlight the `New Content` text string, you can do so using the `strong` element as follows:

```
elem.innerHTML = "<strong>New Content</strong> for the paragraph text";
```

To prevent a *cross-site scripting* attack, HTML5 provides that a `<script>` tag inserted in `innerHTML` must not be executed. Because you can still run JavaScript without a `script` element, you shouldn't use `innerHTML` for strings that are beyond your control.

19.5.2 Changing the Value of an HTML Attribute

You can also change the value of HTML attributes. First you need to search for the corresponding HTML element as usual and change the relevant attribute in it. Here's a simple example where you change the `src` and `alt` attributes of an `img` element to change the picture. First, the HTML document:

```
...
<h1>Change picture</h1>
<p><img class="pic" src="images/bild01.jpg" old="image-01" ></p>
<button onclick="changePicture()">Change picture</button>
<script src="js/script.js"></script>

...
```

Listing 19.20 /examples/chapter019/19_5_2/index.html

And here's the JavaScript for it:

```
let xchange = true;

function changePicture() {
    let current = document.querySelector('.pic');

    if (xchange) {
        current.src = "images/image-02.png";
        current.old = "JavaScript Handbook";
        xchange = false;
    } else {
        current.src = "images/image-01.png";
        current.old = "HTML Handbook";
        xchange = true;
    }
}
```

Listing 19.21 /examples/chapter019/19_5_2/index.html

In this example, you start the JavaScript function `changePicture()` every time the user clicks the **Change picture** button. First, you search the HTML document for an element with `class="pic"` using `querySelector()` and assign the element to the `current` variable.

The `if` check and `else` branch are only needed to enable you to exchange the image again and again when the button has been clicked by checking the global variable `xchange` and setting it again in the corresponding branch according to the change. The actual modification of the `src` and `old` attributes takes place with `current.src` and `current.old`, respectively, and the assignment of a different image and text, respectively. You can also change the value of an attribute in one go as follows:

```
document.querySelector('.pic').src = "images/image-02.png";
document.querySelector('.pic').old = "JavaScript Handbook";
```

The principle works exactly as shown here with all other HTML attributes. Let's take a look at the example in Figure 19.11.

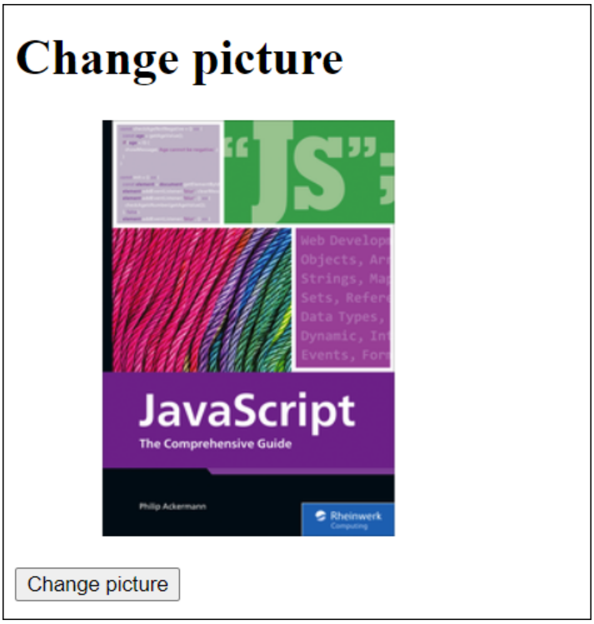


Figure 19.11 In This Example, the "src" and "old" Attributes of the <img> Element Are Changed So the Image Gets Replaced

19.5.3 Changing the Style (CSS) of an HTML Element

Changing the style of an HTML element is relatively easy once the relevant element has been found. The following example changes the style of the `p` element. The `h1` element can be changed by clicking the button, whereupon the self-written JavaScript function `changeColor()` gets called and does its work. Here's the corresponding basic HTML structure:

```
...
<h1 class="headline">Change HTML style</h1>
<p class="p-style">A simple paragraph text ...</p>
```

```
<button onclick="changeColor()">Change color</button>
<script src="js/script.js"></script>

...
```

Listing 19.22 /examples/chapter019/19_5_3/index.html

Here's the corresponding JavaScript code:

```
let element = document.querySelector('.p-style');
element.style.color = "navy";
element.style.background = "snow";
element.style.font = "1.2em Arial";

function changeColor() {
    let headline = document.querySelector('.headline');
    headline.style.color = "gray";
    headline.style.font = "2.5em serif";
    headline.style.fontStyle = "italic";

}
```

Listing 19.23 /examples/chapter019/19_5_3/js/script.js

In this example, you first search for the HTML element (here, via `querySelector()`) and change the corresponding HTML element via `.style.cssProperty` by assigning a valid CSS value. The `p` element was given a blue font color (`color="navy"`) and a snow-white background (`background="snow"`). In addition, the font size was set to `1.2em` and the font to `Arial`.

Camel Case

You may be a little puzzled about `fontStyle` in the */examples/chapter019/19_5_3/js/script.js* example and wonder why `font-style` wasn't used, which is the actual name of the CSS property. In JavaScript, the hyphens are used for the minus sign, which is why the hyphen isn't used here, and the first letter after a hyphen is capitalized. A CSS property such as `border-color` becomes `borderColor` in JavaScript, or `border-top-right-radius` becomes `borderTopRightRadius`. This notation for CSS properties is referred to as a *camel case*.

The style of the `h1` element, on the other hand, doesn't change until the user clicks the button (`onclick` event). When the button is clicked, the font color of the `h1` element will be formatted to gray, and the font style will be italic via `2.5em` and a serif font. You can see the example during execution in Figure 19.12.

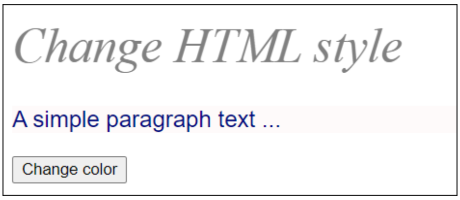


Figure 19.12 Changing the Style of an HTML Element

19.6 Responding to JavaScript Events

You can make your websites truly interactive using *JavaScript events*. Typical examples include the changing of images when the mouse is hovering over an HTML element or checking an input in forms.

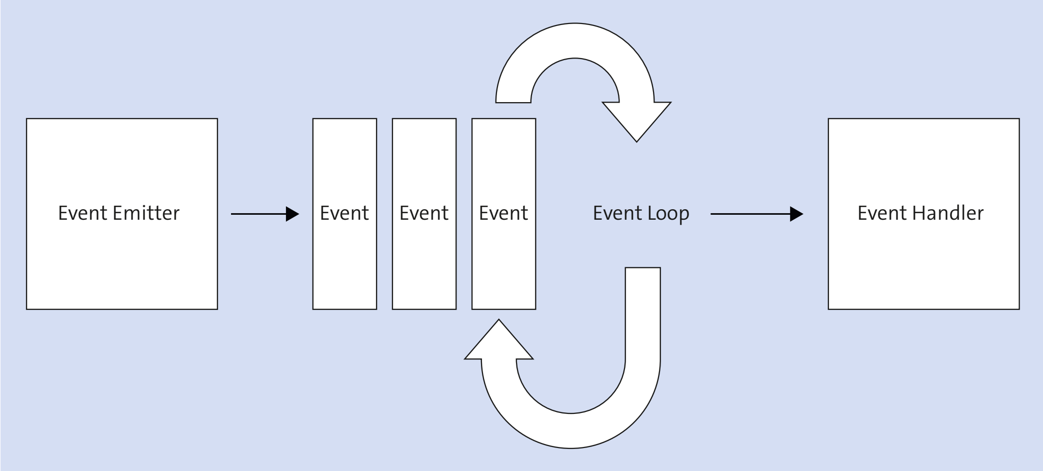


Figure 19.13 Classic Principle of Events and Event Handling

The principle is relatively simple: In the web browser, an *event* gets triggered when an action is performed in the document, in the web browser, or on a specific HTML element. For example, the web browser generates an event when the web page is fully loaded, the mouse is moved, or a button gets clicked. The triggered event is then enqueued in an event queue to ensure that an event which was triggered first is also handled first—the classic first in, first out principle. An event loop continuously checks whether a new event is present in the event queue and passes the event to an event handler. If you're interested in a specific event here, you can register an event handler function for it, which will be executed when an event of that type gets triggered.

You don't need to worry about the *event* types themselves, that is, which event has occurred. These are already included in JavaScript and applicable via special keywords. The number of available events in JavaScript is quite extensive, and you'll first learn about only the most common ones in this section. JavaScript provides events to the following areas:

- User interface (UI) events of the window (e.g., `onload`, `onunload`, `onresize`, `onscroll`, `onerror`, and `onabort`)
- Mouse events (e.g., `onclick`, `ondblclick`, `onmousedown`, `onmousemove`, `onmouseover`, `onemouseout`, and `onmouseup`)
- Keyboard events (e.g., `onkeypress`, `onkeydown`, and `onkeyup`)
- Form events (e.g., `onblur`, `onchange`, `onfocus`, `onreset`, `onselect`, and `onsubmit`)
- Touch events (e.g., `touchstart`, `touching`, `touchcancel`, `touchleave`, and `touchmove`)
- Events for playing video and audio (e.g., `onplay`, `oncanplay`, `onpause`, `oncanplaythrough`, `onplaying`, `ondurationchange`, `onvolumechange`, and `onended`)
- Drag-and-drop events (e.g., `ondrag`, `ondragend`, `ondragenter`, `ondragleave`, `ondragover`, `ondragstart`, and `ondrop`)
- Animation events for CSS animations (e.g., `animationstart`, `animationend`, `animationcancel`, and `animationiteration`)

As you can see, the range of event types is enormous. In this chapter, you'll only learn about classic events of a UI, such as mouse and keyboard events.

In addition to the type of event, you need an *event target* to which the event is linked. For example, if you want to handle a `click` event for a button, you must also use that button as the event target, which is usually a `button` element.

In addition to the event target that's to be monitored and upon which you want to respond to an event, and the event type, you also need a *callback function*, which is supposed to be called when the event for the event target has occurred. This callback function can be set up using an *event handler* for a specific object and event type. Strictly speaking, a distinction is made here between event handlers, which are defined via the properties (e.g., `onclick`), and event listeners, which are defined via the `addEventListener()` method. The difference is that only one event handler can be defined on an element per event, but multiple event listeners can be defined.

When an event of the specified type is triggered at the specified target, the web browser calls the event handler so that you can take care of it with a JavaScript bounce function and respond accordingly. When the event handler is called for an object, this also often refers to the web browser having "triggered" the event.

19.7 Handling the Events Using the Event Handler

An *event handler* is a JavaScript statement or function that gets executed when a specific JavaScript event is triggered (or fired). It's relatively easy to set up an event handler for an event, and you have three options for doing so, which I'll describe in the following three sections.

19.7.1 Setting Up an Event Handler as an HTML Attribute in the HTML Element

You've already seen several times in this chapter how you can set up an event handler as an HTML attribute. The principle is simple and clear, and the script can be called when the HTML document gets loaded, if the HTML element has already been loaded.

In the following line, event handler `changeColor()` has been set up for HTML element `<button>` for event `onclick`. This means that when the button gets clicked (`onclick`), function `changeColor()` will be called.

```
...
<button onclick="changeColor()">Change color</button>
...
<script src="js/script.js"></script>
...
```

Listing 19.24 /examples/chapter019/19_5_3/index.html

```
function changeColor() {
    ...
}
```

Listing 19.25 /examples/chapter019/19_5_3/js/script.js

However, the disadvantage of this method is that only one event handler can be registered. The `addEventListener()` method, on the other hand, can be used to register multiple event listeners.

19.7.2 Setting Up Event Handlers as a Property of an Object

To intercept the event, you need an HTML element that catches the event and assigns the event handler to this element. In practice, this could look as follows:

```
...
// Element which should catch the event
let element = document.querySelector('selector');
// Event to be intercepted and event handler
element.onmouseover = function() { ... };
...
```

Because functions are full-fledged objects in JavaScript, it's no problem to assign them to a property as we did here.

You can write the entire process in one go as follows:

```
document.querySelector('selector').onmouseover = function() { ... };
```

Here's a real-world example in which an event handler is set up as a property of an object:

```
...
<h1>Change HTML style</h1>
<p class="p-style">A simple paragraph text ...</p>

<button id="button01">Change color</button>
<script src="js/script.js"></script>

...
```

Listing 19.26 /examples/chapter019/19_7_2/index.html

```
document.querySelector('#button01').onclick = function() {
    document.querySelector('.p-style').style.color = "navy";
    document.querySelector('.p-style').style.font = "1.2em Arial";
}
```

Listing 19.27 /examples/chapter019/19_7_2/js/script.js

In this example, you assign a function as an event handler to an HTML element where the value of `id` is equal to `button01` (again, the button in this case). This event handler will be executed when the button gets clicked (`onclick`). As an effect, the font color, size, and type of the `p` element are changed in this example. The event handler can be removed again using `element.onclick=null;`. Each object can thus be assigned only one event handler for a specific event. If you add another event handler to an event, the previous event handler will be overwritten.

19.7.3 Setting Up an Event Handler via "addEventListener()"

Similar to assigning the event handler as a property of an object, you can use the `addEventListener()` method to associate an HTML element with an event handler when a specific event occurs. Unlike assigning the event handler directly as a property to an object, you can use the `addEventListener()` method to add an event handler to an HTML element without overwriting an event you had already linked. This allows you to add multiple event handlers to an element and the same event.

Here's an example that demonstrates the `addEventListener()` method:

```
...
<h1>Change HTML style</h1>
<p class="p-style">A simple paragraph text ...</p>

<button id="button01">Change color</button>
```

```
<script src="js/script.js"></script>

...
```

Listing 19.28 /examples/chapter019/19_7_3/index.html

```
let element = document.querySelector('#button01');
if (element) {
    element.addEventListener("click", changeColor);
    element.addEventListener("click", changeText);
    element.addEventListener("mouseover", myborder);
    element.addEventListener("mouseout", noborder);
} else {
    console.log("Error: Could not set up event handler!")
}

function changeColor() {
    document.querySelector('.p-style').style.color = "navy";
    document.querySelector('.p-style').style.font = "1.2em Arial";
}

function changeText() {
    document.querySelector('.p-style').innerHTML = "New Text";
}

function myborder() {
    document.querySelector('.p-style').style.border = "1px solid black";
}

function noborder() {
    document.querySelector('.p-style').style.border = "0px solid black";
}
```

Listing 19.29 /examples/chapter019/19_7_3/js/script.js

Admittedly, I went a bit overboard and assigned four event handlers to the HTML element with `id="btton01"`, that is, the button. Two event handlers will be executed when the button gets clicked (`click` event), and one event handler will be executed each when the user keeps the mouse cursor on the button (`mouseover`) and when the user leaves the button again with the mouse cursor (`mouseout`).

You'll notice here that the prefix `on` is no longer used in the `addEventListener()` method. For example, instead of `onclick`, only `click` is used.

Anonymous Wrapper Function

If you need an event handler with a parameter, you can write an anonymous *wrapper* function, for example:

```
...
function changeText(newTxt) {
  document.querySelector('.update').innerHTML = newTxt;
}
...
let txt = "Text was changed";
element.addEventListener("click", () => {changeText(txt)});
...
```

Here, if the `click` event gets triggered at the element associated with `element`, you can call the `changeText()` function with the `txt` parameter inside an anonymous function.

If you want to remove an event handler from the list, you can do so by using the `removeEventListener()` method as follows:

```
...
element.removeEventListener("click", changeColor);
...
```

19.8 Overview of Common JavaScript Events

You've already learned that there's a large set of JavaScript events for which you can set up an event handler for elements in the HTML document. This section provides a brief overview of the various events.

19.8.1 The JavaScript Events of the UI (Window Events)

Table 19.4 contains an overview of common JavaScript events that are primarily related to the web browser UI and that you can respond to with an event handler. After listing these events, I'll describe how you can use `onload` and `onunload` separately.

JavaScript Event	The Event Occurs When . . .
`error`	An error occurred while a document or image was loaded.
`load`	The document or frameset has been loaded.
`resize`	The document window has changed in size, that is, has been enlarged or reduced.

Table 19.4 Events in the Web Browser UI

JavaScript Event	The Event Occurs When ...
`scroll`	The document was scrolled.
`unload`	The web browser removes a document from the window or frameset (or the web browser gets closed).

Table 19.4 Events in the Web Browser UI (Cont.)

Using the JavaScript Events "onload" and "onunload"

The `onload` event occurs when an object has been loaded, so it's common to use it in the `body` element to execute a script when the entire web page has been loaded with images, scripts, CSS, and so on. You'll often find the following line in the `body` element:

```
<body onload="haveACookie()">
```

Once the entire web page has been loaded, the event handler `haveACookie()` will be executed. Besides that, there's a pure JavaScript version that you can write as follows:

```
...
  window.onload=function() {
    // JavaScript code
  }

...
```

The counterpart of the `onload` event is the `onunload` event, which can be used when a user leaves the page, for example, by clicking a link, submitting a form, or closing the window in the web browser. The `onunload` event is also called when the user reloads the web page. However, in that case, the `onload` event will also be executed. Again, the usage is mainly in the `body` element, and the syntax is also the same as for the `onload` event:

```
<body onload="haveACookie()" onunload="setACookie()">
```

A JavaScript version can be used as follows:

```
...
  window.onunload=function() {
    // JavaScript code
  }

...
```

"DOMContentLoaded" versus "load"

I've already briefly mentioned the `DOMContentLoaded` event once in this book. The event gets triggered once the complete DOM tree of a web page has been loaded. You're probably wondering what the difference is between the events `load` and `DOMContentLoaded`. As described previously, you can define an event listener as follows, for example:

```
function initJS() {
  // JavaScript code
}
window.addEventListener('load', initJS);
```

You can do almost the same with the `DOMContentLoaded` event as follows:

```
function initJS() {
  // JavaScript code
}
document.addEventListener('DOMContentLoaded', initJS);
```

The difference between the two versions is that with `load`, the event is only triggered when the complete document (the DOM tree), including all external resources such as JavaScript files, CSS files, or images, has been loaded. With the `DCOMContentLoaded` event, on the other hand, the event gets triggered if only the entire DOM tree of the web page has been loaded. In your daily work, I recommend you use `DOMContentLoaded` instead of `load` because it avoids waiting for external resources to load.

19.8.2 JavaScript Events That Can Occur in Connection with the Mouse

Many events can occur in connection with the mouse. Table 19.5 contains an overview of the different mouse events.

JavaScript Event	The Result Occurs When . . .
`click`	An element has been clicked.
`dblclick`	An element has been double-clicked.
`mousedown`	The mouse button is pressed over an element.
`mousemove`	The mouse pointer is placed over an element and is then moved.
`mouseover`	The mouse pointer is over the element.
`mouseout`	The mouse pointer gets moved away from an element.
`mouseup`	The pressed mouse button is released again.

Table 19.5 Various Mouse Events to Which You Can Respond

Here's a simple example that demonstrates some of the mouse events in action:

```
...
<h1>Mouse events</h1>
```

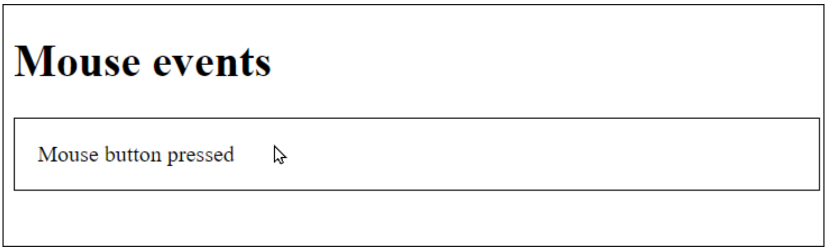
```
<p class="p-style">Mouse pointer here</p>
<script src="js/script.js"></script>
...
```

Listing 19.30 /examples/chapter019/19_8_2/index.html

```
let element = document.querySelector('.p-style');
element.addEventListener("mouseover", mymouseover);
element.addEventListener("mousedown", mymousedown);
element.addEventListener("mouseup", mymouseup);
element.addEventListener("mouseout", mymouseout);

function mymouseover() {
    element.innerHTML = "Mouse pointer over HTML element";
}

function mymousedown() {
    element.innerHTML = "Mouse button pressed";
}

function mymouseup() {
    element.innerHTML = "Mouse button released";
}

function mymouseout() {
    element.innerHTML = "Exit HTML element";
}
```

Listing 19.31 /examples/chapter019/19_8_2/js/script.js

In this example, different event handlers have been set up for the `mouseover`, `mousedown`, `mouseup`, and `mouseout` events for the HTML element with `class="p-style"` using the `addEventListener()` method. Depending on which mouse event setup with an event handler is currently occurring, a note about it will display.



Mouse events

Mouse button pressed

Figure 19.14 Responding to Events

19.8.3 JavaScript Events for Devices with a Touchscreen

Besides mouse events, you also need to consider touch events because an increasing number of devices are controlled with touchscreens instead of a mouse. The `click` event is the same for both mouse operation and touchscreens, and it can be used for both a mouse click on the desktop and the touchscreen on mobile devices. All other mouse events can also be used by mobile browsers on touch devices. Thus, you probably won't need to use special touch events in most cases.

If you write mobile-only applications, it makes sense to replace mouse events with touch events. For an overview of JavaScript events for devices with touchscreens, take a look at Table 19.6. As an alternative, you can use a library that directly supports all events and use the relevant event depending on the end device.

JavaScript Event	The Event Occurs When . . .
`touchstart`	The surface of the touchscreen gets touched. Corresponds to a `mousedown`.
`touchend`	The finger is lifted from the surface of the touchscreen. Corresponds to a `mouseup`.
`touchcancel`	The finger leaves the area.
`touchmove`	The finger slides across the touchscreen's surface. Corresponds to a `mousemove`.

Table 19.6 Various Events for Devices with Touchscreen

More Information Online

If you want to deal specifically with the processing of touch events on mobile devices, this isn't as trivial as it may seem. I won't go into a comprehensive description here. But you can find more information about touchscreens in an excellent documentation at *https://developer.mozilla.org/en-US/docs/Web/API/Touch_events*.

19.8.4 JavaScript Events That Occur in Connection with the Keyboard

You can also respond to keyboard events such as pressing, holding down, and releasing keys. Strictly speaking, when a key is pressed and released, three JavaScript events get triggered in the order `keydown` (press), `keypress` (keep pressed), and `keyup` (release). Table 19.7 contains an overview of the existing keyboard events.

JavaScript Event	The Event Occurs When . . .
`keydown`	A key on the keyboard gets pressed.

Table 19.7 Various Keyboard Events to Which You Can Respond

JavaScript Event	The Event Occurs When . . .
`keyup`	A depressed key of the keyboard gets released.
`keypress`	A key gets pressed and held down.

Table 19.7 Various Keyboard Events to Which You Can Respond (Cont.)

19.8.5 JavaScript Events for HTML Forms

For the HTML forms I covered in Chapter 7, you can also find the available events in Table 19.8.

JavaScript Event	The Event Occurs When . . .
`blur`	A form element loses its focus.
`change`	The content of a form element such as `<input>`, `<select>`, or `<textarea>` has changed.
`focus`	An element such as `<label>`, `<input>`, `<select>`, `<textarea>`, or `<button>` gets the focus.
`reset`	The form gets reset.
`select`	A text has been marked in an input field such as `<input>` or `<textarea>`.
`submit`	The form gets submitted.

Table 19.8 Events That Can Occur in Connection with HTML Forms

19.8.6 JavaScript Events for the Web APIs

Due to the large number of Web application programming interfaces (APIs) that exist, there are also many more events. Especially for playing audio and video files with the `<audio>` and `<video>` elements, you’ll find an extensive list of event types. In addition, the new drag-and-drop API for dragging and dropping elements defines event types. Furthermore, there are various events for the Web APIs such as for the creation of offline web applications or for asynchronous communication.

19.9 More Information about Events with the “event” Object

What you’ve probably wondered about JavaScript events like keyboard events (e.g., a keystroke) is whether it’s somehow possible to determine which key triggers an event. Yes, this is possible, and this information is even already provided as a property of the respective event.

Events are themselves objects that have a variety of properties and methods, which is a typical feature of JavaScript. More precisely, the `event` object gets passed to the function to be called. To access it, you must explicitly pass the `event object` to the function as the first parameter. Here's a simple example that demonstrates this process:

```
...
<h1>Properties of events</h1>
<p onmousedown="showPos(event)" class="p-style">
  Click here!
</p>
<p>Keystroke: <input type="text" onkeydown="keyPressed(event)"></p>
<output></output>
<script src="js/script.js"></script>

...
```

Listing 19.32 /examples/chapter019/19_9/index.html

```
function showPos(ev) {
    let x = ev.clientX;
    let y = ev.clientY;
    let text = "Pos-X: " + x + " / Pos-Y: " + y;
    if (ev.shiftKey == true) {
        text += " / (Shift) key was pressed!";
    } else {
        text += " / (Shift) key was not pressed!";
    }
    text += " -> Mouse button: " + ev.button;
    document.querySelector('output').innerHTML = text;
    console.log(ev);
}

function keyPressed(ev) {
    let text = "Key code: " + ev.keyCode + "=
" + String.fromCharCode(ev.keyCode);
    document.querySelector('output').innerHTML = text;
}
```

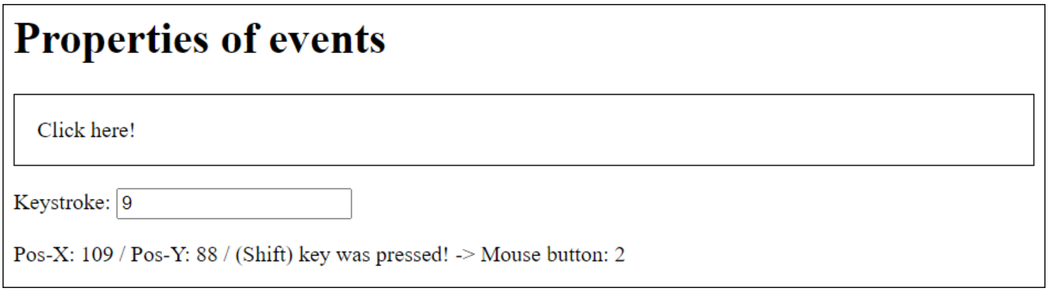
Listing 19.33 /examples/chapter019/19_9/js/script.js

You can see here how the `event` object is used as a parameter for the event handler to establish a link to the event's properties. Within the function, you can access the values or properties of the `event` object, which was done here via `clientX` and `clientY`, returning the horizontal and vertical coordinate position, respectively, relative to the current window in which the corresponding event (in the example: `onmousedown`) was triggered.

Likewise, it was tested here whether the `shiftKey` property equals `true`, which means that the `Shift` key was pressed at the same time the mouse was pressed.

In addition, `console.log(ev)` provides a convenient way to get all the information about the `event` object output to the JavaScript console, which is why you should open the JavaScript console.

I then set up something similar for the keyboard by using `onkeydown`, where the `event` object is used to output the keyboard code (`keyCode`) and, if possible, the corresponding character for it.



Properties of events

Click here!

Keystroke: 9

Pos-X: 109 / Pos-Y: 88 / (Shift) key was pressed! -> Mouse button: 2

Figure 19.15 The “event” Object Can Also Be Used to Access Further Information about an Event

Besides the `clientX`, `clientY`, or `shiftKey` properties of an `event` object presented in the example, there are many more properties, some of which you can find listed in Table 19.9.

Property	Description
`altKey`, `ctrlKey`, `shiftKey`	Returns whether or not the `Alt`, `Ctrl`, or `Shift` key was held down when the event occurred.
`bubbles`	Indicates whether an event can ascend or not—more precisely, whether an event also applies to the parent element and can ascend in the DOM tree. This is referred to as *bubbling*.
`button`	Contains a value that indicates which mouse button was pressed. `0` stands for the left, `1` for the center, and `2` for the right mouse button.
`clientX` `clientY`	Contains the horizontal (`clientX`) and vertical (`clientY`) pixel value with the mouse cursor position relative to the upper-left corner of the window.
`cancelable`	Returns whether the default action of an event can be prevented or not.
`currentTarget`	Returns the element whose event listener triggered the event.

Table 19.9 Overview of Some Properties of JavaScript Events

Property	Description
`keyCode`	Contains the keyboard code of the last key pressed. You can determine the character using `String.fromCharCode(event.keyCode)`.
`metaKey`	Indicates whether the meta key was pressed when the event occurred. In practice, this is the `cmd` key on a Mac.
`screenX` `screenY`	Contains the horizontal (`screenX`) and vertical (`screenY`) pixel value with the mouse pointer position relative to the upper-left corner of the screen.
`target`	Returns the element that triggered the event.
`type`	Returns the name of the event.

Table 19.9 Overview of Some Properties of JavaScript Events (Cont.)

19.10 Suppressing the Default Action of Events

HTML elements such as a simple link or a button to submit a form perform the intended actions when events occur, even without JavaScript. For example, clicking on a link triggers a `click` event, which usually causes the web browser to load a new web page. Clicking a **Submit** button on a form also triggers a `submit` event, which submits the form to the web server.

Certain events, as just mentioned, are handled by the web browser in such a way that it executes a *default action* without the need to set up a special event handler function in JavaScript.

Here's a simple example that demonstrates this:

```
<a class="aLink" href="https://www.rheinwerk-computing.com/">A link</a>
```

Now, if you want to take control of JavaScript yourself and, instead of following the link and loading a new web page, do something else, you need to suppress the default action of the element. For this purpose, you can use the `preventDefault()` method of the `event` object:

```
event.preventDefault();
```

You can use this method to suppress the intended default action. This process is also referred to as an *event cancellation*.

Here's a complete example that demonstrates how you can suppress the default action of the `click` event for a link. In the example, only the default action of following the link and loading the website is suppressed, and a message gets displayed instead in the console indicating that the default action has been suppressed:

```
...
document.querySelector('.aLink').onclick = function(event) {
    if (event.preventDefault) {
        event.preventDefault();
    }
    // Write the actual JavaScript code here, which states
    // what is supposed to happen when the link gets clicked on
    console.log("Default action prevented");
}
...
```

Listing 19.34 /examples/chapter019/19_10/js/script.js

19.11 The Event Flow (Event Propagation)

Based on what you know so far, you register an event handler for a specific event and target element that you want to monitor and respond to. If the corresponding event occurs on the target element, the event handler will be triggered and executed. However, this description of the event flow is a bit vague. For this reason, we'll briefly describe the event flow at this point, that is, how an event moves through the DOM tree when an event gets triggered. This event flow is also referred to as *event propagation* and takes place in three phases:

1. **Capturing phase**
   The event descends from the topmost document node in the DOM tree to the target element of the event. The phase starts with the intercepting handlers of the `window` object, through those of the `document` object and the `body` object, down to those of the event target's parents. Due to this descent to the target element, it becomes possible to look more closely at the events before they reach their destination. For example, an intercepting event handler can be used to suppress certain events, as we did earlier in the chapter with the `preventDefault()` method.
2. **Target phase**
   When the event reaches its target element, the corresponding event handlers get triggered (i.e., on the target element), which were registered for the corresponding event type, and the bubbling phase is initiated.
3. **Bubbling phase**
   Starting from the target element, the event now rises back up the element hierarchy of the DOM tree. This rising is called *bubbling*. It ensures that the registered handlers of the target element are also executed in its parent element, its grandparent element, and so on up to the `document` object and then the `window` object. In other words, the event triggered on the target element rises up the DOM tree, where it triggers the event handlers/event listeners registered for that event on those elements.

19.11.1 More about the Bubbling Phase

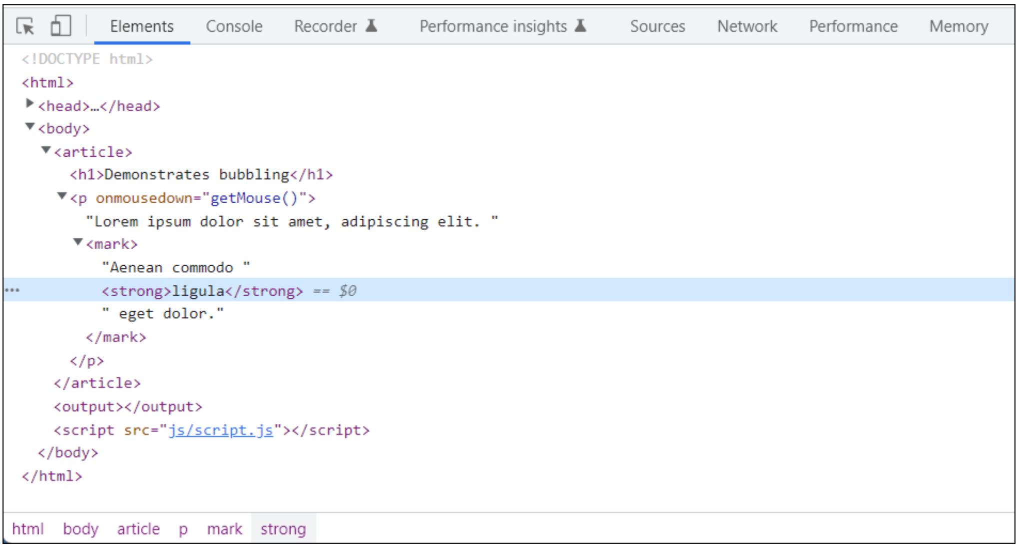
It may sound relatively pointless at first that in the capturing phase, everything in the DOM tree initially descends from the top down to the target element, only to ascend back up again in the bubbling phase. If you look at the following HTML code and the explanation that follows, you'll quickly understand the meaning of bubbling:

```
...
  <article>
    <h1>Demonstrates bubbling</h1>
    <p onmousedown="getMouse()">Lorem ipsum dolor sit amet, adipiscing elit.
      <mark>Aenean commodo <strong>ligula</strong> eget dolor.</mark>
    </p>
  </article>
  <output></output>

  <script>

    function getMouse() {
      var text = "Mouse button was pressed!"
      var pos = document.querySelector('output');
      if (pos) {
        pos.innerHTML = text;
        pos.style.background = "lightgray";
      }
    }

  </script>
...
```

Listing 19.35 /examples/chapter019/19_11_1/index.html

In the example, the handler function `getMouse()` has been set up for the `p` element in the `article` element, which is executed when you press the mouse button in the `p` element. If you press the left mouse button in the `p` element, corresponding information is output in the `output` element. The same thing happens if you click on the `mark` element within the `p` element or on the `strong` element one level further down in the hierarchy. This occurs due to bubbling. For example, if the event is triggered on the `mark` element, the event will rise, and the handler in the `p` element will be executed. This way, you can avoid registering handlers for individual elements and instead register a single handler for a common ancestor object and respond to the event there.

While most event types ascend via the bubbling phase, there are some that don't go up and trigger a corresponding handler in the ancestor object. For some event types such as the `focus`, `blur`, and `scroll` events, ascending in this way wouldn't be very useful.


Elements
Console
Recorder
Performance insights
Sources
Network
Performance
Memory

```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<article>
      <h1>Demonstrates bubbling</h1>
     ▼<p onmousedown="getMouse()">
        "Lorem ipsum dolor sit amet, adipiscing elit. "
       ▼<mark>
          "Aenean commodo "
...       <strong>ligula</strong> == $0
          " eget dolor."
        </mark>
      </p>
    </article>
    <output></output>
    <script src="js/script.js"></script>
  </body>
</html>
```

html
body
article
p
mark
strong

Figure 19.16 Thanks to Bubbling, the Event That Triggers on the <strong> Element or <mark> Element Will Rise, Which Will Execute the Handler of the <p> Element

19.11.2 Canceling Bubbling via the "stopPropagation()" Method

The process of rising in the bubbling phase isn't always desired and can therefore be prevented via the `stopPropagation()` method. Take a look at the following HTML code, where bubbling is more annoying than useful:

```
...
<article onmousedown="getMouseArticle()">
  <h1>Demonstrates bubbling</h1>
  <p onmousedown="getMouseP()">Lorem ipsum dolor ... </p>
</article>
<script src="js/script.js"></script>

...
```

Listing 19.36 /examples/chapter019/19_11_2/index.html

```
let text = "";
function getMouseP() {
    alert("Mouse button was pressed in p!");
}
function getMouseArticle() {
    alert("Mouse button was pressed in article!");
}
```

Listing 19.37 /examples/chapter019/19_11_2/js/script.js

Here, two nested elements—the `p` element and `article` element—monitor the same event (`mousedown`). If you click on the `p` element inside the `article` element, the handler function `getMouseP()` will be executed. Because of bubbling, the event rises to the top and, upon reaching the `article` element, executes the `getMouseArticle()` handler function registered for the same event (`mousedown`) there as well.

You can stop this rising of nested elements by using the `stopPropagation()` method. Here's the corresponding example:

```
...
<article onmousedown="getMouseArticle()">
  <h1>Demonstrates bubbling</h1>
  <p onmousedown="getMouseP(event)">Lorem ipsum dolor ... </p>
</article>
<script src="js/script-2.js"></script>

...
```

Listing 19.38 /examples/chapter019/19_11_2/index-2.html

```
function getMouseP(ev) {
    alert("Mouse button was pressed in p!");
    if (ev.stopPropagation) {
        ev.stopPropagation();
    }
}

function getMouseArticle() {
    alert("Mouse button was pressed in article!");
}
```

Listing 19.39 /examples/chapter019/19_11_2/js/script-2.js

When the left mouse button in the `p` element gets pressed, the `getMouseP(ev)` handler function set up for this purpose will be called with the `event` object as a parameter. In the handler function, it's first checked whether the `stopPropagation` method exists, and if it does, it will be executed. By using `stopPropagation()`, you've stopped bubbling for the event type `mousedown`, and the type no longer gets passed up to the `article` element.

19.11.3 Intervening in the Event Flow during the Capturing Phase

If you want to intervene in the event flow in the capturing phase, that is, when it runs from top to bottom, you can only do that by using the `addEventListener()` method. Using `<element onevent="handlerFunc()">` as HTML attribute (Section 19.7.1) or `element.onevent = handlerFunc` as a property of an object (Section 19.7.2), you can register the handler only for the bubbling phase. In the description of `addEventListener()` in Section 19.7.3, I withheld from you the third parameter where you can specify the `Boolean` value `true` or `false`. If you set the value to `true` in that parameter, you register the event handler or event listener for the capturing phase. By default, the third parameter is set to `false` and therefore doesn't need to be explicitly specified in that case either.

Here's an example that shows how you can set up a handler for the capturing phase:

```
...
  <p class="my-p">Lorem ipsum dolor... </p>
  <script src="js/script.js"></script>

...
```

Listing 19.40 /examples/chapter019/19_11_3/index.html

```
let id = document.querySelector('.my-p');
if (id) {
    id.addEventListener("mousedown", getMouseP, true);
} else {
    alert("No element with id=myId found!");
}

function getMouseP() {
    alert("Mouse button was pressed in p!");
}
```

Listing 19.41 /examples/chapter019/19_11_3/js/script.js

This way, you can edit nonascending events already on an ancestor element, for example. In addition, if you call the `stopPropagation` method of the `event` object already in the capturing phase, the target and bubbling phases won't get executed at all, and thus the event type won't reach its target element.

19.11.4 Additional Information on the Capturing and Bubbling Phases

Now that you know the difference between the descending capturing phase to the target element and the ascending bubbling phase, here's some more useful information. If nonascending event types such as `load`, `focus`, `blur`, `mouseenter`, `mouseleave`, or `submit` get triggered, an intervention in the capturing phase via `addEventListener("eventtype", handler, true)` enables you to monitor such types at a central position and thus process the higher-level elements during their descent. Unlike the bubbling phase, you can be sure that every event has a capturing phase.

In addition, if you register an event handler with `addEventListener()` for the capturing phase and that handler was called in that phase, you can be sure that it won't be called again in the bubbling phase. For this reason, a registration can only be made for the capturing phase or the bubbling phase, not for both at the same time.

19.12 Adding, Changing, and Removing HTML Elements

The term *node* comes up quite often in connection with the DOM and always causes confusion. The principle is simple: At the top of the DOM tree, there's the `document` object, followed by the root of the DOM tree, which is usually the `html` element. The `head` and `body` elements follow below that. The `html` element is the parent element for the `head` and `body` elements. The `head` and `body` elements contain other child elements. You already know the principle. All these elements are referred to as *nodes* in the DOM. Not only are the HTML elements nodes for the JavaScript in the DOM, but the HTML attributes and text contents are also connected and are real nodes in the DOM tree.

Regardless of the node type, all nodes in the document tree have basic properties and methods of the `node` object. This `node` object is an interface that allows you to access the individual nodes in the entire document tree, so it's the central interface in the DOM. Therefore, in this section, you'll get to know some basic properties and methods of the `node` interface.

So far, you've primarily only learned about and used manipulation via `innerHTML`. This is relatively easy in practice, but it has the disadvantage that an element must be present that can be replaced or changed. In the following sections, you'll learn how to use the DOM methods that allow you to directly intervene in the DOM of an HTML document using the `node` interface.

You'll learn how to navigate through the individual nodes of the tree. Likewise, you'll learn how to add new nodes (HTML elements) to the tree or remove or replace existing nodes. In particular, this includes DOM manipulation.

At the beginning of this chapter (Section 19.1), you already learned about the HTML DOM tree and that all nodes in a tree have a certain relationship to each other. You also know from Section 19.2 that the `document` object is always the topmost object in the DOM tree. The `document` object is also a kind of start element where you can begin to target the nodes of a DOM tree. For this purpose, the `document` object provides the `getElementById()`, `getElementsByTagName()`, `getElementsByClassName()`, `getElementsByName()`, `querySelector()`, and `querySelectorAll()` methods that allow you to search for elements by ID, a specific tag name, a class name, a CSS selector, or a `name` attribute. I already covered this in Section 19.4.

We should also mention two options that allow you to access the entire document:

- **`document.body`**
  Access to the entire `body` element in the HTML document.
- **`document.documentElement`**
  Access to the entire HTML document, that is, the `head` and `body` parts.

Paste the following into your HTML document to see what's contained in `document.body` and/or `document.documentElement`:

```
...
<script>
  console.log(document.body.innerHTML);
  console.log(document.documentElement.innerHTML);
</script>
...
```

19.12.1 Creating and Adding a New HTML Element and Content

Creating a new node or HTML element isn't difficult. To do this, you just need to create a new HTML element in the `document` object using the `createElement()` method. Text content, on the other hand, can be created using the `createTextNode()` method. A new node created in this way is initially not yet connected to the HTML document. You still need to explicitly `append` the created node to the DOM tree using `appendChild()`. Here's a simple example, which I'll explain in more detail afterward:

```
...
<body>
<article class="article-01">
  <h1>Add node</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>

<button id="new-p">New paragraph</button>
<script src="js/script.js"></script>


</body>
...
```

Listing 19.42 /examples/chapter019/19_12_1/index.html

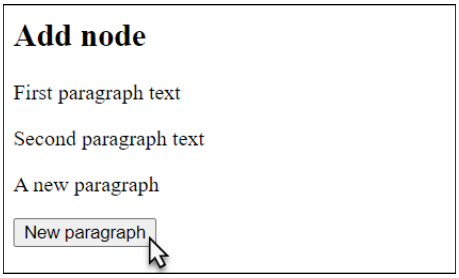
```
document.querySelector('#new-p').onclick = function() {
    let pNew = document.createElement("p");
    let tNew = document.createTextNode("A new paragraph");
    pNew.appendChild(tNew); // <p>A new paragraph</p>

    document.querySelector('.article-01').appendChild(p_new);
}
```

Listing 19.43 /examples/chapter019/19_12_1/js/script.js

When you click the button in the example, you create a new `p` element via `createElement("p")`. Then you create another text content with `createTextNode()`, and position the text content into the `p` element with `p_new.appendChild(tNew)`, creating the following:

```
<p>A new paragraph</p>
```

Finally, you need to paste the entire construct in `pNew` into the HTML document. Therefore, in this example, you look for an element where the value of `class` is equal to `article-01`, and append the created element with `appendChild(pNew)` in the `article` element after the second paragraph text as the third paragraph, as you can see in Figure 19.17.



Add node

First paragraph text

Second paragraph text

A new paragraph

Figure 19.17 A Newly Created Paragraph Text Was Added

19.12.2 Targeting HTML Elements Even More Exactly in the DOM Tree

To do more than just look up the position in the DOM tree based on a root element and append newly created elements at the end, as you did in the previous section, you need to be able to navigate through the DOM tree. For this purpose, several DOM properties are available, which are listed in Table 19.10.

Property	Description
`parentNode`	Returns the parent node. This access is useful when an element can be found unambiguously, but not the parent element.
`childNodes[n]`	Returns an array with all child nodes.
`firstChild`	Returns the first child node.

Table 19.10 Properties You Use to Navigate through the DOM Tree

Property	Description
`lastChild`	Returns the last child node.
`nextSibling`	Returns the next node on the same level, that is, the sibling node.
`previousSibling`	Returns the preceding node on the same level (sibling node).

Table 19.10 Properties You Use to Navigate through the DOM Tree (Cont.)

With these properties, you could already dare a small climb in the DOM tree, but it's not really reliable yet. Because, for example, line breaks are also read as new nodes and are virtually regarded as text elements within an HTML element, you can't avoid checking the nodes. For this purpose, Table 19.11 contains additional properties that allow you to analyze the individual nodes.

Property	Description
`nodeType`	Returns the type of the node; it's arguably one of the most important analysis functions when navigating and manipulating the DOM tree due to different DOM implementations of web browsers. A numerical code from `1` to `12` is returned, of which the values `1` and `3` are most frequently required: `1` gets returned for an element node and `3` for a text node. There are also constants defined in the DOM API for the values you can use instead: `1: ELEMENT_NODE`, `2: ATTRIBUTE_NODE`, `3: TEXT_NODE`, `4: CDATA_SECTION_NODE`, `7: PROCESSING_INSTRUCTION_NODE`, `8: COMMENT_NODE`, `9: DOCUMENT_NODE`, `10: DOCUMENT_TYPE_NODE`, `11: DOCUMENT_FRAGMENT_NODE` The values and constants for 5, 6, and 12 are obsolete and will therefore not be mentioned any further here.
`nodeName`	Returns the name of the node as a string. For an HTML element, this is a tag name (usually capitalized), whereas for an attribute it's the attribute name. The text node simply returns `#text`, and the document node returns `#document`.
`nodeValue`	Returns the content of a text node (e.g., `innerHTML`) or the value of the attribute node. If the node is an HTML element, the value is `undefined`.
`hasChildNodes`	Allows you to check whether a node has other child nodes (= `true`) or not (= `false`).

Table 19.11 Properties for Analyzing Nodes

Here's a somewhat more complex example that searches for a node, traverses the included individual elements, and outputs their properties:

```
...
<body>
<article class="article-01">
  <h1>Article 1: Traverse node</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
...
<script src="js/script.js"></script>


</body>
...
```

Listing 19.44 /examples/chapter019/19_12_2/index.html

```
let root = document.querySelector('.article-01');
if (root) {
    let traverse = root.childNodes;
    let text = traverse.length + " Elements are " +
        root.nodeName + " contained in:" + "<ol>";
    for (let i = 0; i < traverse.length; i++) {
        text += "<li>" + "<b>nodeName</b>: " + traverse[i].nodeName +
            "; <b>nodeType</b>: " + traverse[i].nodeType;
        if (traverse[i].firstChild !== null) {
            text += "; <b>nodeValue:</b> " + traverse[i].firstChild.nodeValue;
        }
        text += '</li>';
    }
    text += "</ol>" + "Parent node: " + root.parentNode.nodeName;
    document.querySelector('#result').innerHTML = text;
} else {
    alert("No child nodes available!!!");
}
```

Listing 19.45 /examples/chapter019/19_12_2/js/script.js

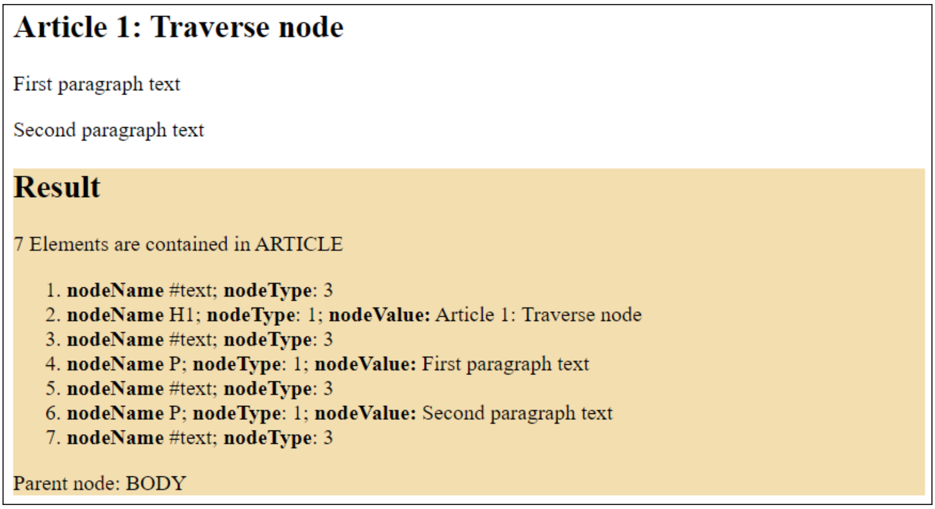
After first checking whether the root node with the `class` of value `article-01` could be found at all, you pass a list of child nodes `root.childNodes` to the `traverse` variable if successful. The first information you put into the `text` string is the number of elements found and the name of the root node. In the subsequent `for` loop, all nodes are traversed and the information of each node gets appended to the string `text`.

Checking a `traverse[i].firstChild` node for `non-zero` was done because a line break is also considered a text node, and we don't really want that information. Otherwise, if it isn't a line break, you'll find the contents of the text node in `traverse[i].firstChild.nodeValue`.

You can see the example during execution in Figure 19.18, and you'll notice that a total of seven nodes were found where the parent element had the `class` attribute that equals `article-01`. Look a little closer at the following HTML lines:

```
...
<article class="article-01">
  <h1>Article 1: Traverse node</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
...
```

You'll notice that there should actually only be three nodes. The nodes where `nodeName` equals `#text` and the value of `nodeType` equals `3` are again line breaks. You can see that it's very important to check the nodes to see if they are an element node or a text node. In practice, you can use `getElementsByTagName()` for this, which usually returns element nodes.



Article 1: Traverse node

First paragraph text

Second paragraph text

Result

7 Elements are contained in ARTICLE

1. **nodeName** #text; **nodeType**: 3
2. **nodeName** H1; **nodeType**: 1; **nodeValue:** Article 1: Traverse node
3. **nodeName** #text; **nodeType**: 3
4. **nodeName** P; **nodeType**: 1; **nodeValue:** First paragraph text
5. **nodeName** #text; **nodeType**: 3
6. **nodeName** P; **nodeType**: 1; **nodeValue:** Second paragraph text
7. **nodeName** #text; **nodeType**: 3

Parent node: BODY

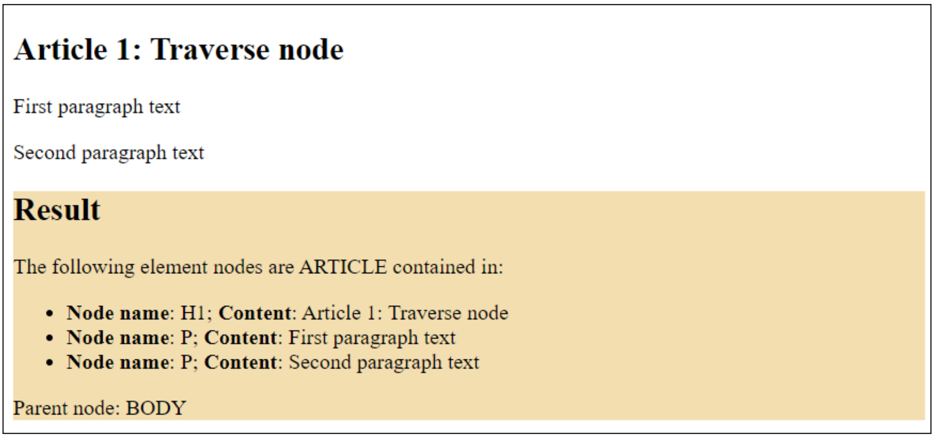
Figure 19.18 Traversing a Root Node

You can entirely omit such empty line breaks from the analysis when going through the individual elements:

```
let root = document.querySelector('.article-01');
if ( root ) {
  let traverse = root.childNodes;
  let text = "The following element nodes are "
           + root.nodeName  + " contained in:" + "<ul>";
  for (let i=0; i<traverse.length; i++) {
    if (traverse[i].firstChild !== null ) {
      text += "<li>" + "<b>Node name</b>: " + traverse[i].nodeName + "; "
           + "<b>Content</b>: " + traverse[i].firstChild.nodeValue + "</li>";
    }
  }
  text += "</ul>" + "Parent node: " + root.parentNode.nodeName;
  document.querySelector('#result').innerHTML = text;
}
else {
  alert("No child nodes available!!!");
}
```

Listing 19.46 /examples/chapter019/19_12_2/js/script-2.js

You can see the changed example during execution in Figure 19.19.



Article 1: Traverse node

First paragraph text

Second paragraph text

Result

The following element nodes are ARTICLE contained in:

- **Node name**: H1; **Content**: Article 1: Traverse node
- **Node name**: P; **Content**: First paragraph text
- **Node name**: P; **Content**: Second paragraph text

Parent node: BODY

Figure 19.19 Rewritten Version for Traversing the Root Node, Which Doesn't Take into Account the Line Breaks in the Output in the Dialog Box

19.12.3 Adding a New HTML Element Even More Targeted to the DOM Tree

You can use the `appendChild()` method to add a new node as the last child node of the root element. If you want to insert the new node at any position, the `insertBefore(new, old)` method is useful, which inserts a `new` node before `old` below the root node.

The following example shows how you can insert a new node (here, a new paragraph text) right after an `h1` heading using the `insertBefore()` method:

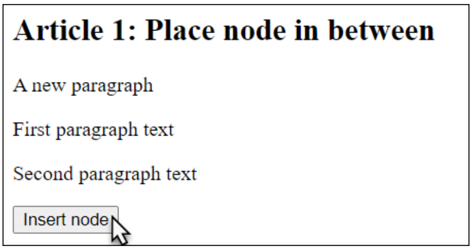
```
...
<article class="article-01">
  <h1>Article 1: Place node in between</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
<button id="add">Insert node</button>
<script src="js/script.js"></script>

...
```

Listing 19.47 /examples/chapter019/19_12_3/index.html

```
document.querySelector('#add').onclick = function() {
    let pNew = document.createElement("p");
    let tNew = document.createTextNode("A new paragraph");
    p_new.appendChild(tNew);

    let root = document.querySelector('.article-01');
    if (root) {
        let traverse = root.childNodes;
        for (let i = 0; i < traverse.length; i++) {
            if (traverse[i].nodeName.toUpperCase() === "H1") {
                root.insertBefore(pNeu, traverse[i].nextSibling);
                break; // end loop
            }
        }
    }
}
```

Listing 19.48 /examples/chapter019/19_12_3/js/script.js

First, you look for the root node again by using `class="article-01"`. Then, in a `for` loop, you go through the `child` nodes of the root node you've found. If you then find a node in `nodeName` that contains the `h1` element, you insert the newly created node (here, `pNew`) after the `h1` element. To make sure that the new paragraph doesn't get inserted before the `h1` heading, you must also use `nextSibling`, which positions the new node as the next node (i.e., after the `h1` element) on the same layer. You can see the example during execution in Figure 19.20.



Article 1: Place node in between

A new paragraph

First paragraph text

Second paragraph text

Figure 19.20 Positioning the New Node in a Targeted Manner. Here, a New <p> Element Was Inserted after the <h1> Heading

19.12.4 Deleting an Existing HTML Element from the DOM Tree

If you want to remove an already existing child node from the DOM tree, you can do this by using the `removeChild(child)` method. The node to be deleted can be an entire fragment with further nodes, which also deletes all subnodes, or a child node standing alone. In the example, when a button is clicked, the first paragraph with the `p` element is supposed to be deleted.

```
...
<article class="article-01">
  <h1>Article 1: Remove node</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
<script src="js/script.js"></script>

...
```

Listing 19.49 /examples/chapter019/19_12_4/index.html

```
document.querySelector('#remove').onclick = function() {
    let root = document.querySelector('.article-01');
    if (root) {
        let traverse = root.childNodes;
        for (let i = 0; i < traverse.length; i++) {
            if (traverse[i].nodeName.toUpperCase() === "P") {
                root.removeChild(traverse[i]);
                break; // end loop
            }
        }
    }
}
```

Listing 19.50 /examples/chapter019/19_12_4/js/script.js

Here, you first search for the root node using `class="article-01"` and then traverse its child nodes. In this process, you remove the `p` element via `removeChild()` right after the first occurrence. If, on the other hand, you want to delete all `p` elements at once, you only need to remove the `break` statement in the example.

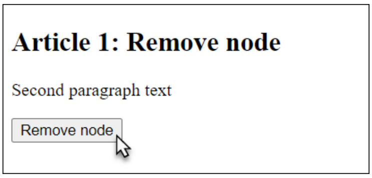


Figure 19.21 A <p> Element Has Been Removed

19.12.5 Replacing an HTML Element in the DOM Tree with Another One

You can replace a node by using the `replaceChild(new, old)` method, which replaces the `old` node with the `new` node. In the process, the replaced node gets deleted. Again, you can replace entire fragments with many child nodes or a standalone child node. In the following example, each time the button is clicked, the heading gets replaced. To keep track of how many times this replacement has been performed or how many times the button has been clicked, a global variable is used that counts and is included in the heading when the replacement is made. Here's the corresponding example:

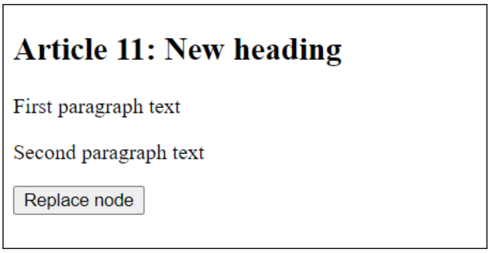
```
...
<article class="article-01">
  <h1>Article 1: Replace node</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>

<button id="replace">Replace node</button>
<script src="js/script.js"></script>

...
```

Listing 19.51 /examples/chapter019/19_12_5/index.html

```
let counter = 0;
document.querySelector('#replace').onclick = function() {
    let hNew = document.createElement("h1");
    let tNew =
 document.createTextNode("Article " + ++counter + ": New heading");
    hNew.appendChild(tNew); // <p>A new paragraph</p>.

    let root = document.querySelector('.article-01');
    if (root) {
        let traverse = root.childNodes;
        for (let i = 0; i < traverse.length; i++) {
            if (traverse[i].nodeName.toUpperCase() === "H1") {
                root.replaceChild(h_new, traverse[i]);
                break; // end loop
            }
        }
    }
}
```

Listing 19.52 /examples/chapter019/19_12_5/js/script.js

You can see the example during execution in Figure 19.22.



Article 11: New heading

First paragraph text

Second paragraph text

Figure 19.22 Replacing Nodes: The Heading Has Already Been Changed for the 11th Time

19.12.6 Cloning a Node or Entire Fragments of the DOM Tree

If you want to copy an entire fragment of a DOM tree, you're better off using the `cloneNode()` method than creating individual elements via `createElement()`. You can use the `cloneNode()` method to create an exact copy of the node. Depending on whether you set the parameter of `cloneNode(val)` to `true` or `false`, all child nodes will either be cloned as well (= `true`) or not (= `false`). Here's an example that shows how an entire fragment can be cloned:

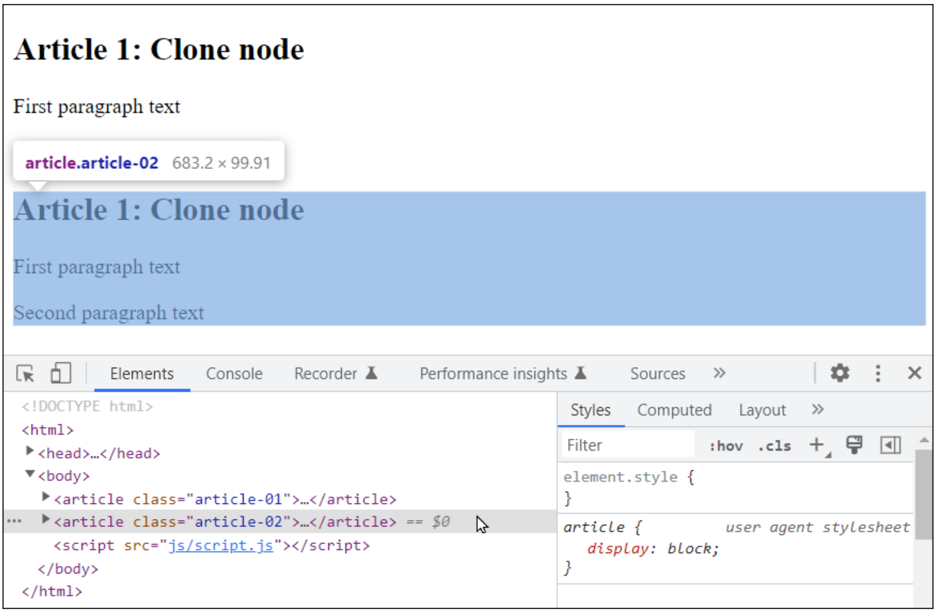
```
...
<article class="article-01">
  <h1>Article 1: Clone node</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
<button id="clone">Clone node</button>
<script src="js/script.js"></script>

...
```

Listing 19.53 /examples/chapter019/19_12_6/index.html

```
document.querySelector('#clone').onclick = function() {
    let root = document.querySelector('.article-01');
    if (root) {
        let new_root = root.cloneNode(true);
        new_root.setAttribute("class", "article-02");
        let button = document.querySelector('#clone');
        root.parentNode.insertBefore(new_root, button);
        button.parentNode.removeChild(button);
    }
}
```

Listing 19.54 /examples/chapter019/19_12_6/js/script.js

You can see the example during execution in Figure 19.23. Here, you clone the entire node whose `class` is equal to `article-01`, which is the `article` element, along with its child nodes `<h1>` and the two `<p>` elements. Before you can place the node somewhere else, in this example, the `class` was changed to `article-02` with the method `setAttribute()` because, otherwise, you would have two elements with `class="article-01"`, which isn't wrong but something you might not want. This depends on your specific project, of course, but I wanted to demonstrate the `setAttribute()` method here. In the example, the cloned node gets inserted after the previous article and before the button. The button got deleted at the end.



Article 1: Clone node

First paragraph text

Article 1: Clone node

First paragraph text

Second paragraph text

```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▶<article class="article-01">…</article>
…  ▶<article class="article-02">…</article> == $0
    <script src="js/script.js"></script>
  </body>
</html>
```

```
element.style {
}
article {                user agent stylesheet
   display: block;
}
```

Figure 19.23 A Second Article Was Cloned from the First Article

19.12.7 Different Methods to Manipulate the HTML Attributes

As you've already seen in the previous section with the `setAttribute()` method, it's possible to change the attributes of the nodes. Table 19.12 contains a brief overview of the existing methods, of which `setAttribute()` and `getAttribute()` are probably most commonly used in practice.

Method	Description
`getAttribute(name)`	Returns a string with the value of the `name` attribute. If no such attribute exists, `null` will be returned.

Table 19.12 Methods That Can Be Used to Work with the Attributes of Nodes

Method	Description
`setAttribute(name,value)`	This method sets the `name` attribute to the value `value`. If an attribute with `name` already exists in an element, the value will be changed to `value`. If the attribute doesn't exist yet, it will be created.
`removeAttribute(name)`	Removes the `name` attribute from an element.
`hasAttribute(name)`	Checks whether an element node contains the `name` attribute.

Table 19.12 Methods That Can Be Used to Work with the Attributes of Nodes (Cont.)

Here's an example that demonstrates these methods in practice.

```
.demo {
    background: black;
    color: white;
    padding: 1em;
    margin-bottom: 1em;
    border: 1px solid red;
}

.default {
    border: 1px solid black;
    background: silver;
    padding: 1em;
    margin-bottom: 1em;
}
```

Listing 19.55 /examples/chapter019/19_12_7/css/style.css

```
...
<article id="article-01" class="default">
  <h1>Article 1: Manipulate attributes</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
<article id="article-02">
  <h1>Article 2: Manipulate attributes</h1>
  <p>First paragraph text</p>
  <p>Second paragraph text</p>
</article>
<button id="set">Set attribute</button>
<button id="copy">Copy attribute</button>
```

```
<button id="remove">Delete attribute</button>
<script src="js/script.js"></script>

...
```

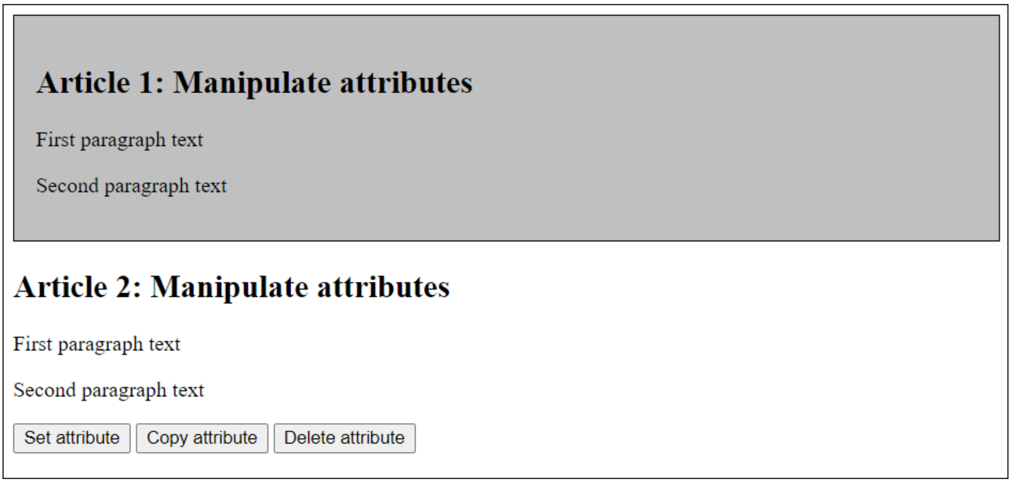
Listing 19.56 /examples/chapter019/19_12_7/index.html

```
document.querySelector('#set').onclick = function() {
    let root = document.querySelector('#article-01');
    if (root) {
        root.setAttribute("class", "demo");
    }
}

document.querySelector('#copy').onclick = function() {
    let root1 = document.querySelector('#article-01');
    if (root1) {
        let art01_style = root1.getAttribute("class");
        if (art01_style != null) {
            let root2 = document.querySelector('#article-02');
            if (root2) {
                root2.setAttribute("class", art01_style);
            }
        }
    }
}

document.querySelector('#remove').onclick = function() {
    let root1 = document.querySelector('#article-01');
    if (root1) {
        root1.removeAttribute("class");
    }
    let root2 = document.querySelector('#article-02');
    if (root2) {
        root2.removeAttribute("class");
    }
}
```

Listing 19.57 /examples/chapter019/19_12_7/js/script.js

Three event handler functions have been written here, all of which are executed when the button gets clicked. The first function changes or sets the class of an article (via `id="article-01"`) to the value `demo` using the `setAttribute()` method and the `class` attribute, resulting in `class="demo"`. The original default `class="default"` is overwritten.

The second function gets the value of the `class` attribute using the `getAttribute()` method of the first article (`id="article-01"`) and sets this value also for the second article (`id="article-02"`) using `setAttribute()`. The last function deletes the `class` attribute for the first (`id="article-01"`) and the second article (`id="article-02"`) via the `removeAttribute()` method. All three functions can be performed using the three buttons shown in Figure 19.24. Of course, you can apply and execute these methods on any other HTML attribute as well.



Article 1: Manipulate attributes

First paragraph text

Second paragraph text

Article 2: Manipulate attributes

First paragraph text

Second paragraph text

Figure 19.24 Manipulating the Attributes of an Element Node

19.12.8 The <template> HTML Tag

For recurring repetitions on a website, such as lines of a table, list elements, or images, templates are often used. The processing of such templates was mostly done on the server side. PHP programmers use template systems such as Plates or Twig for this purpose. Because JavaScript itself doesn't provide a template system, it's often necessary to create and assemble HTML elements by using `createElement()`. A solution to this is the `template` HTML element. For example:

```
<!-- Rows template -->
<template id="template-row">
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</template>
```

All elements between the `<template>` tag don't display and aren't part of the HTML document. In addition, this element can't be accessed via `document.getElementById()`. The content of the `<template>` tag won't get used until the fragment has been added to the DOM.

The following example demonstrates the implementation using a table by adding new rows to a table via the `<template>` tag:

```
...
<h1>Scheduling overview</h1>
<table id="mytable">
<thead>
<tr>
<td>Time</td>
<td>Day</td>
<td>Date</td>
</tr>
</thead>
<tbody>

<!-- The rows are added here -->

</tbody>
</table>

<!-- Rows template -->
<template id="template-row">
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</template>
...
```

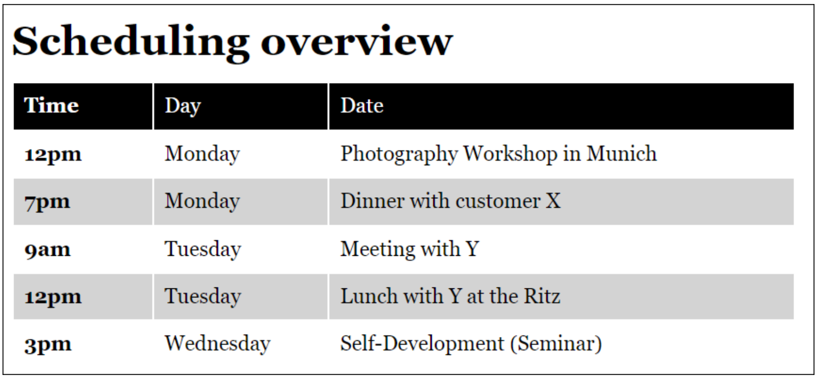
Listing 19.58 /examples/chapter019/19_12_8/index.html

So, without further ado, the `template` element of a table row with three columns is inactive, and nothing gets displayed yet. To use this template now, you need to clone it and add it to the DOM. The corresponding JavaScript code follows:

```
...
<script>
let data = [
   ['12pm','Monday','Photography Workshop in Munich'],
   ['7pm', 'Monday', 'Dinner with customer X'],
   ['9am', 'Tuesday', 'Meeting with Y'],
   ['12pm', 'Tuesday', 'Lunch with Y at the Ritz'],
   ['3pm', 'Wednesday', 'Self-Development (Seminar)']
];

let t = document.querySelector('#template-row');
td = t.content.querySelectorAll('td');

data.forEach (function(dataRow) {
   td[0].textContent = dataRow[0];
   td[1].textContent = dataRow[1];
   td[2].textContent = dataRow[2];

   let tb = document.querySelector('tbody');
   let clone = document.importNode(t.content, true);
   tb.appendChild(clone);
});
</script>
...
```

Listing 19.59 /examples/chapter019/19_12_8/index.html

The data for the rows in the table here comes from the `data` array with five rows and three columns. Then `querySelector()` is used to select the template with the `template-row` ID, and thereupon all `td` elements are put into `td`. Via `data.forEach()`, the data in `data` is processed row by row and inserted column by column as child elements in the `tbody` element. You can see the result of this process in Figure 19.25.



Scheduling overview

Time	Day	Date
12pm	Monday	Photography Workshop in Munich
7pm	Monday	Dinner with customer X
9am	Tuesday	Meeting with Y
12pm	Tuesday	Lunch with Y at the Ritz
3pm	Wednesday	Self-Development (Seminar)

Figure 19.25 The Data of the Table Was Inserted into the DOM Using the <template> Element and JavaScript

19.13 HTML Forms and JavaScript

Although HTML input types and attributes now do most of the work that you would have had to do with JavaScript some time ago, there are always reasons to get your hands dirty to check entries that have been made or options selected. For this reason, this section will describe how you can access the individual values of a form element using JavaScript.

Basically, evaluating form elements with JavaScript is quite simple. If a certain event occurs, you can respond to it accordingly. For example, with a single-line input field, you can respond appropriately when the field loses focus and the user moves to the next field. To do that, you only need to respond to `onblur` using an event handler. With regard to selection lists, on the other hand, reading only makes sense if they've been changed, which is why you can respond to `onchange` with an event handler and with multiline text fields. Radio buttons and checkboxes are usually grouped, so you can respond with an event handler for a `click` event (`onclick`), and then you should check all elements that belong to the group.

19.13.1 Reading Text Input Fields with JavaScript

You can get the value of a text input field such as `<input type="text">` directly from the `value` attribute. To respond to changes, an event handler is useful for the JavaScript event `onblur`, which gets triggered when the input field loses its focus. This is the case when the user clicks on another input field or changes the field using the Tab key. That's the moment to use JavaScript and check the text input field. Let's take a look at a simple example:

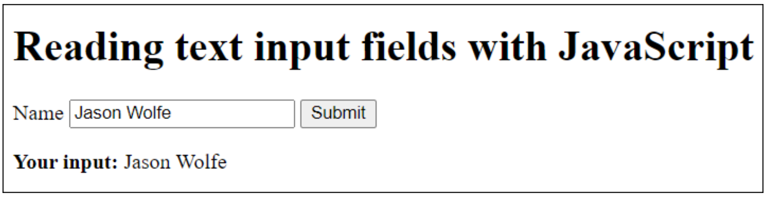
```
...
<form>
  <label>Name</label>
  <input type="text" placeholder="Your name" id="lname"><br>
  <input type="submit">
</form>
<p></p>
<script src="js/script.js"></script>
...
```

Listing 19.60 /example/chapter019/19_13_1/index.html

```
document.querySelector('#lname').onblur = function() {
    let txt = "<b>Your input:</b> " + this.value;
    // Check saved value of text field in txt
    document.querySelector('p').innerHTML = txt;
};
```

Listing 19.61 /Beispiel/Kapitel019/19_13_1/js/script.js

In this example, if you enter text in the input field with `lname` as the ID and press the Tab key or click outside the text input field, causing the input field to lose focus, the entered text gets output below it. In practice, you perform a check at this point or continue processing the input that has been entered instead of just outputting the text here. `this` is the object that just called the JavaScript function.



Reading text input fields with JavaScript

Name Jason Wolfe

Your input: Jason Wolfe

Figure 19.26 Reading the Content of an “input” Input Field “type="text"” with JavaScript

19.13.2 Reading Selection Lists with JavaScript

For the `select` selection lists, you also get the value of the `option` element via the HTML attribute `value`. By default, the text between `<option>` and `</option>` or else that of the `value` attribute in the opening `<option>` tag will get returned if it was used instead. Here, it’s useful to respond to the JavaScript event `onchange`, which responds when the `select` selection list changes. Here’s another short example:

```
...
<form>
  <label>Your selection</label>
  <select id="chapter">
    <option value> ... Select chapter ... </option>
    <option>HTML5-Introduction</option>
    <option value="Page 324">CSS3-Introduction</option>
    <option value="Page 498">JavaScript-Introduction</option>.
  </select>
</form>
<p></p>
<script src="js/script.js"></script>
...
```

Listing 19.62 /examples/chapter019/19_13_2/index.html

```
document.querySelector('#chapter').onchange = function() {
    let txt = "<b>Your selection:</b> " + this.value;
    // Check saved value of text field in txt
    document.querySelector('p').innerHTML = txt;
};
```

Listing 19.63 /examples/chapter019/19_13_2/js/script.js

In this example, as soon as you change the value of the dropdown list (`onchange`), the new value between `<option>` and `</option>` or, if specified, the value of the `attribute` value in the opening `<option>` tag, will be output underneath it in the `p` element.

"onchange" Also for <textarea>

The JavaScript event `onchange` is usually also used for multiline text between `<textarea>` and `</textarea>` to check if the content has been changed once the `textarea` element has lost its focus.

19.13.3 Reading Radio Buttons and Checkboxes with JavaScript

The reading of the values of a radio button (`type="radio"`) and a checkbox (`type="checkbox"`) must be evaluated using the JavaScript event `onclick`. For each input field of a group with radio buttons or a group with checkboxes, you need to check whether it was selected via a `click` event.

A related group of radio buttons or checkboxes usually has a common `name` attribute and can also be addressed in a group using JavaScript. Unlike radio buttons, checkboxes can have more than one field selected, so here you must use JavaScript and the `checked` property to verify that a field in the group is active or checked.

Here's an example that demonstrates how you can use JavaScript to determine the values of a group of radio buttons or checkboxes when something has been changed:

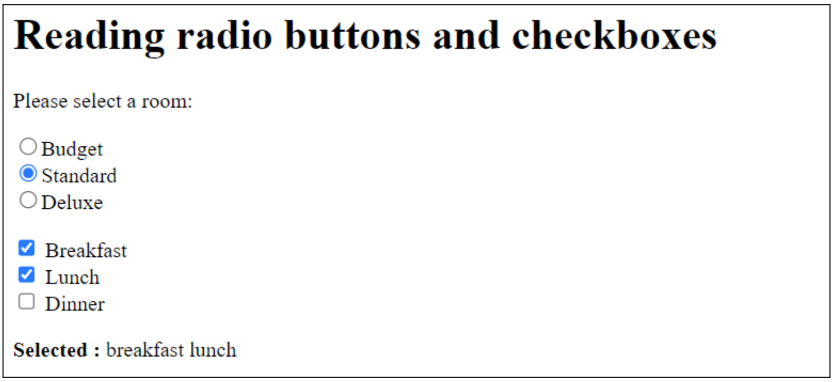
```
...
<form>
  <p>Please select a room:</p>
  <p>
    <input type="radio" name="room" value="budget">Budget<br>
    <input type="radio" name="room" value="standard"
           checked>Standard<br>
    <input type="radio" name="room" value="deluxe">Deluxe
  </p>
  <p id="printRoom"></p>
  <p>
    <input type="checkbox" name="extra" id="c1"
           value="breakfast">
    <label for="c1">Breakfast</label><br>
    <input type="checkbox" name="extra" id="c2" value="lunch">
    <label for="c2">Lunch</label><br>
    <input type="checkbox" name="extra" id="c3" value="dinner">
    <label for="c3">Dinner</label>
  </p>
  <p class="output"></p>
```

```
</form>
<script src="js/script.js"></script>
...
```

Listing 19.64 /examples/chapter019/19_13_3/index.html

```
// Evaluate radio buttons
let roomType = document.querySelectorAll('input[name="room"]');
for (let i = 0; i < roomtyp.length; i++) {
    roomType[i].onclick = function() {
        let txt = "<b>Selected for room : </b>" + this.value;
        document.querySelector('.output').innerHTML = txt;
    }
}

// Evaluate checkboxes
let formPack = document.querySelectorAll('input[name="extra"]');
for (let i = 0; i < formPack.length; i++) {
    formPack[i].onclick = function() {
        let msg = "<b>Selected : </b>";
        for (let j = 0; j < formPack.length; j++) {
            if (formPack[j].checked) {
                msg += formPack[j].value + " ";
            }
        }
        document.querySelector('.output').innerHTML = msg;
    }
}
```

Listing 19.65 /examples/chapter019/19_13_3/js/script.js



Reading radio buttons and checkboxes

Please select a room:

○ Budget
● Standard
○ Deluxe

☑ Breakfast
☑ Lunch
☐ Dinner

Selected : breakfast lunch

Figure 19.27 Reading Radio Buttons and Checkboxes with JavaScript

19.13.4 Intercepting Buttons with JavaScript

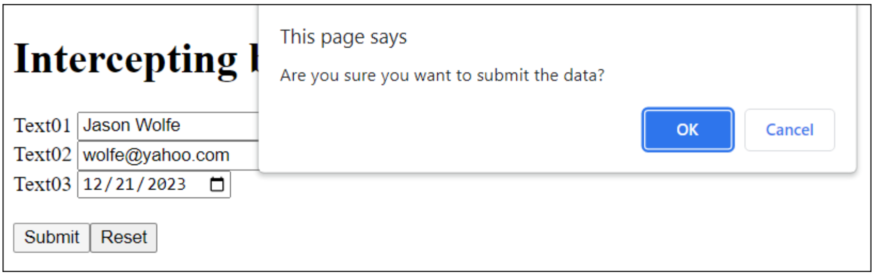
If you want to prevent the user from submitting the entered form data in the web browser to the server by clicking a button, you can use an event listener (Section 19.7.3) to respond to the `submit` event and have another script executed. Let's take a look at a simple example:

```
...
<form>
  Text01 <input type="text"  id="t1">
  <label for="t1"></label><br>
  Text02 <input type="email" id="t2">
  <label for="t2"></label><br>
  Text03 <input type="date" id="t3">
  <label for="t3"></label><br><br>
  <input type="submit"><input type="reset">
</form>
<p></p>
<script src="js/script.js"></script>

...
```

Listing 19.66 /examples/chapter019/19_13_4/index.html

```
document.querySelector('form').addEventListener(
    'submit', checkInput);
document.querySelector('form').addEventListener(
    'reset', checkReset);

function checkInput() {
    let x = confirm("Are you sure you want to submit the data?");
    if (x) {
        /* Submit data */
    } else {
        /* Do not submit */
        event.preventDefault();
    }
}

function checkReset(event) {
    let x = confirm("Do you want to reset the fields?");
    if (x) {
        /* Reset */
        document.querySelector('p').innerHTML = "Fields reset";
    } else {
        /* Do not reset */
        event.preventDefault();
        document.querySelector('p').innerHTML = "Reset canceled";
    }
}
```

Listing 19.67 /examples/chapter019/19_13_4/js/script.js



Intercepting I

This page says

Are you sure you want to submit the data?

Text01 Jason Wolfe
Text02 wolfe@yahoo.com
Text03 12/21/2023

Figure 19.28 You Can Also Respond to “submit” and “reset” Buttons with JavaScript

19.13.5 Controlling the Progress Indicator <progress> with JavaScript

You’ve already come across the `progress` display with the `progress` element in Chapter 7, Section 7.5.5. With reference to an HTML form, this element is useful for showing the progress of form element input. The following example shows such an adjustment of the progress bar in accordance with the filled input fields. Let’s look at a simple example:

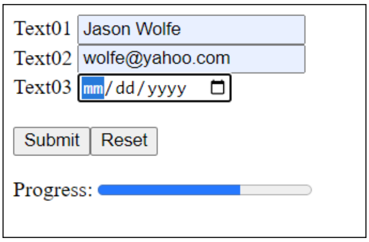
```
...
<form onchange="progress()">
  Text01 <input type="text"  id="t1"><br>
  Text02 <input type="email" id="t2"><br>
  Text03 <input type="date" id="t3"><br><br>
  <input type="submit"><input type="reset">
</form>
<p>Progress: <progress id="bar" value="0" max="3">Progress</progress></p>
<script src="js/script.js"></script>

...
```

Listing 19.68 /examples/chapter019/19_13_5/index.html

```
function progress() {
    let fields = 0;
    for (let i = 0; i < document.forms[0].elements.length; i++) {
        if (document.forms[0].elements[i].value !== '') {
            fields++;
        }
    }
    document.querySelector('#bar').value = fields;
    document.querySelector('#bar').innerHTML="Progress (" + fields + " of 3): ";
}
```

Listing 19.69 /examples/chapter019/19_13_5/js/script.js

The example checks each change in the form (`onchange`) and then calls the handler function `progressive()`. The individual elements will be traversed, and a check will be performed for each element as to whether the content (`value`) is or isn't an empty string. No empty string in the example means that this text field has already been filled in, and therefore the count variable `fields` is incremented with each filled text field. This value is used at the end in the progress bar for `value` of the `progress` element, where we've set the maximum value to `3`. In the example, we also rely on bubbling and don't set up the same handler for the event type `onchange` in each `input` element within the form individually. In Figure 19.29, you can see the example during execution.



Text01 Jason Wolfe

Text02 wolfe@yahoo.com

Text03 mm/dd/yyyy

Figure 19.29 Controlling the Progress Bar from the <progress> Element with JavaScript

19.14 Summary

In this chapter, you've learned a lot about the DOM and DOM manipulation. The abundance of information in such a small space might have been a bit overwhelming. For this reason, here's a brief summary of the most important information you've learned and should know:

- You know how to search for a specific element in a DOM tree starting from the `document` object using the following methods `querySelector()`, `querySelectorAll()`, `getElementById()`, `getElementsByTagName()`, `getElementsByClassName()`, and `getElementsByName()`.
- You know the `innerHTML` property, which you can use to access or modify the contents of an HTML element. In addition, you know other properties or methods that you can use to change the value of an attribute (`attribute`, `setAttribute`) or the style (`style.property`) of an HTML element.

- You know what JavaScript events are and how to respond to and handle such events with event handlers. Likewise, you're now familiar with many common events.
- You've learned how to traverse a DOM tree, adding, modifying, or deleting new elements.
- You've gotten to know the HTML tag `<template>` as a way to implement templates on the client side in HTML and JavaScript.
- You've learned how to use JavaScript to access the individual values of form elements.

Chapter 20

An Introduction to Ajax

If you've studied JavaScript in more detail, you'll inevitably come across the concept of asynchronous data transfer with Ajax. Ajax is an important and useful technology in building websites, so I'll cover the basics here.

To understand this technology, knowledge of JavaScript, HTML, and CSS is required. This chapter describes what Ajax is and how you can use this technology in your projects. Ajax is not a new technology or programming language, but rather a programming concept. Strictly speaking, it is just JavaScript, server calls, and an intervention in the DOM via JavaScript. In short, it's just using existing technologies in a particular way—sounds more complicated than it is!

20.1 An Introduction to Ajax Programming

The often scary and cryptic *Asynchronous JavaScript and XML* (Ajax) is used to transfer data asynchronously between a web browser and a web server.

XML

As if everything weren't already extensive enough, XML is added here as a technology or as a further markup language. Like HTML, XML is a markup language in which the data is hierarchically structured as human-readable text data. XML is widely used for exchanging data between different computer systems, in particular over the internet. Although the *x* in Ajax stands for XML, *JavaScript Object Notation* (JSON) is increasingly used in practice when a web server sends complete data, while the use of XML is waning.

Though all this sounds quite complex, it's actually just a way to refresh individual parts of a web page without reloading the entire web page, which is usually what happens without Ajax. Consequently, you can create faster dynamic web pages by using Ajax. This reduces the amount of data transfer and also saves the nerves of website visitors thanks to shorter loading times.

In the context of Ajax, the term *asynchronous* means that script execution continues when an HTTP request is made, as this request to the web server is executed in the background, and the web page is still available to the user. Usually, that is, without Ajax, such an operation runs *synchronously*, which means that the script execution is paused until the requested data has returned from the web server. There are definitely situations where you need to use a synchronous operation even with Ajax and just do one thing at a time. Figure 20.1 represents a synchronous operation, while Figure 20.2 represents an asynchronous operation.

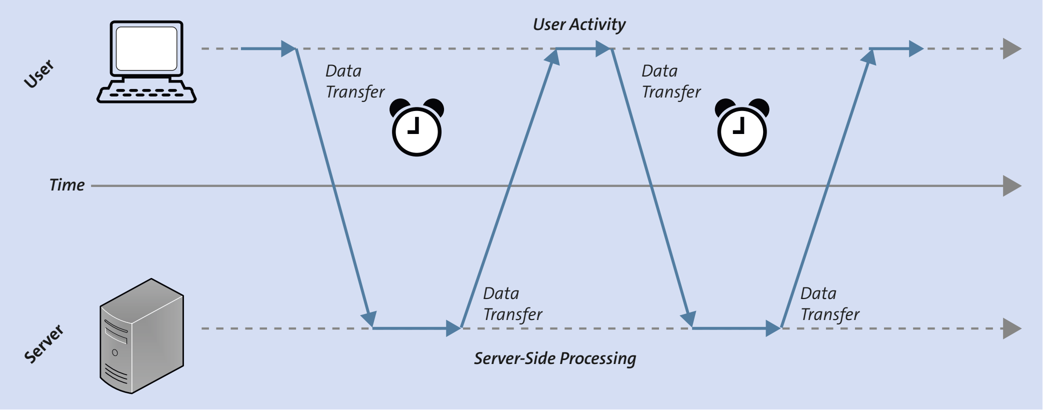

User Activity
User
Data Transfer
Data Transfer
Time
Data Transfer
Data Transfer
Server
Server-Side Processing

Figure 20.1 The Synchronous Process Flow of a Classic Web Application

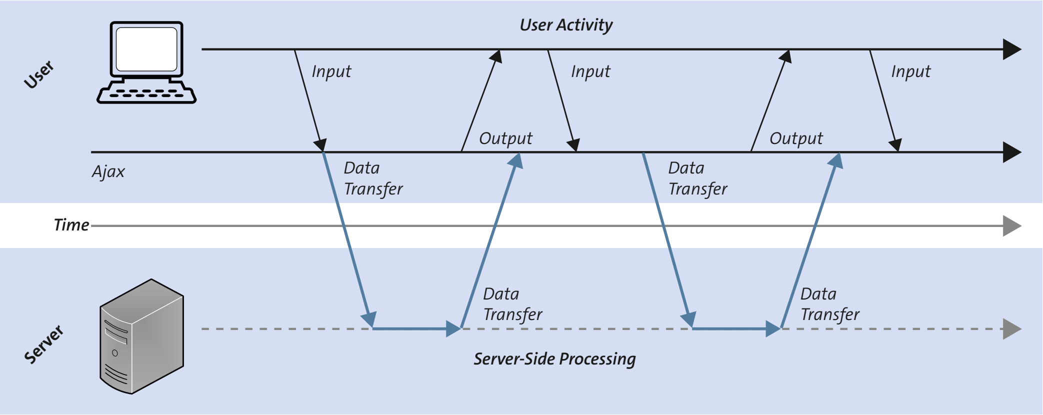

User Activity
User
Input
Input
Input
Output
Output
Ajax
Data Transfer
Data Transfer
Time
Data Transfer
Data Transfer
Server
Server-Side Processing

Figure 20.2 The Asynchronous Process Flow of a Web Application with Ajax

Another advantage of Ajax is that Ajax applications are independent of the web browser and operating system, and just about any web browser can handle them. Simply put, an Ajax application often consists of a combination of the following components:

- The HTTP request with the `XMLHttpRequest` object to exchange the data with the server asynchronously
- JavaScript/DOM for displaying and interacting with the information and data

- XML as the format for data transfer (in addition to XML, methods such as JSON have also become established for asynchronous data transfer)
- CSS for designing the data

20.1.1 A Simple Ajax Example in Execution

It might be easiest to try out a small Ajax example for yourself in practice. In doing so, you'll discover that Ajax isn't rocket science. Because you can't test the example offline, you should run it on a real web server. Either you upload the files to your web host for this purpose, or you've set up your own web server locally. As always, you can test the example online at *https://html-examples.pronix.de/*. For this purpose, here's the complete HTML document shown in Figure 20.3 during execution. I'll go into the details of the example later.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Simple Ajax example</title>.
  <script src="js/getServerTime.js"></script>

</head>
<body onload="timestamp()">
  <p id="refreshtime">
     Ajax is used to output the time of the server.
  </p>
  <button type="button" onclick="changeContent()">
      Renew time
  </button>
  <p id="timestamp"></p>
</body>
</html>
```

Listing 20.1 /examples/chapter020/20_1/index.html

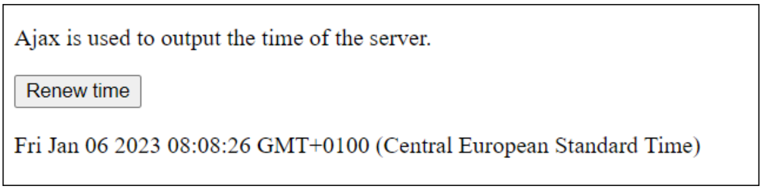
```
function changeContent() {
    let xmlhttp = null;
    if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
    if (xmlhttp === null) {
        console.log("Error creating an XMLHttpRequest object");
    }
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
            document.querySelector('#refreshtime').innerHTML =
 xmlhttp.responseText;
        }
    }
    xmlhttp.open("GET", "php/server-time.php", false);
    xmlhttp.send();
}

function timestamp() {
    let today = new Date();
    document.querySelector('#timestamp').innerHTML = today;
}
```

Listing 20.2 /examples/chapter020/20_1/js/getServerTime.js

Here is the code for the PHP script *server-time.php*, which, in this example, should be placed in the same directory as *index.html*:

```
<?php
  echo date('l jS \of F Y h:i:s A');
?>
```

Listing 20.3 /examples/chapter020/20_1/server-time.php


Ajax is used to output the time of the server.

Fri Jan 06 2023 08:08:26 GMT+0100 (Central European Standard Time)

Figure 20.3 This Is What the Example Looks Like

First, here's the HTML part of the Ajax example:

```
...
<body onload="timestamp ()">
  <p id="refreshtime">
   Ajax is used to output the time of the server.
  </p>
  <button type="button" onclick="changeContent()">
    Renew time
  </button>
  <p id="timestamp"></p>
```

```
</body>
...
```

Listing 20.4 /examples/chapter020/20_1/index.html

The short Ajax example contains a `p` element with the ID `refreshtime` and a `button` button that calls a function named `changeContent()`. After the element with the button, there's another `p` element with the ID `timestamp`. When loading the web page (`onload`), the `timestamp()` function gets executed too, which was written in the `body` element. All in all, the entire thing looks like an ordinary HTML file that wants to use various JavaScript functions as event handlers.

20.1.2 Creating the "XMLHttpRequest" Object

The key to an Ajax application is the use of an `XMLHttpRequest` object. All modern web browsers are capable of doing this. You need this `XMLHttpRequest` object to exchange data with a web server and thus refresh individual parts of a web page without having to reload the entire web page. Here's the section with the creation of the `XMLHttpRequest` object:

```
...
function changeContent() {
  let xmlhttp;

  if (window.XMLHttpRequest) {
    xmlhttp = new XMLHttpRequest();
  }

...
}

...
```

Listing 20.5 /examples/chapter020/20_1/js/getServerTime.js

Before you can create an `XMLHttpRequest` object, you should first verify that the web browser does actually support `XMLHttpRequest` objects. All modern web browsers will create a new `XMLHttpRequest` object using the new syntax `variable=new XMLHttpRequest()`.

20.1.3 Making a Request to the Server

With the generated `XMLHttpRequest` object, you can make a request to the server to exchange data with it. To make such a request, you need to connect to the target page using the `open()` method and specify the parameters of the `XMLHttpRequest` object via `send()`. In the preceding example, you made this request to the server with the following lines:

```
...
  xmlhttp.open("GET", "server-time.php", true);
  xmlhttp.send();
...
```

Listing 20.6 /examples/chapter020/20_1/js/getServerTime.js

The `open()` method has the following syntax:

```
open(method, url, async)
```

You can use `method` to specify the method of the request (HTTP request method), which is usually `GET` or `POST`.

You use `url` to specify the path or URL to the file on the server to be accessed. By the way, this can be any file, and it doesn't necessarily have to be an executable script on the server, like a PHP script in our example.

You can use `async` to specify whether the request should be executed asynchronously (`true`) or synchronously (`false`). In practice, the asynchronous transfer is usually recommended, that is, setting the value to `true`, which is what Ajax—Asynchronous JavaScript and XML—stands for. The advantage of an asynchronous data transfer with Ajax is that a JavaScript no longer has to wait for the server's response and can meanwhile execute other scripts or process the response if it's available. With a synchronous transfer, on the other hand, a JavaScript doesn't get executed any further until the response from the server is available. If the server is slow or busy, the application would stop with `async=false` and remain in a waiting state until the response from the server arrives. Here's a short summary:

- `async=true` stands for *asynchronous* and makes sure that the script continues to run while the HTTP request is being executed in the background.
- `async=false` stands for *synchronous* and causes the script execution to stop until the data has returned from the server.

The `send()` method, on the other hand, allows you to send the request along with the data to the server.

20.1.4 Sending Data

Although in this example, no data gets sent to the script, it should be briefly mentioned here how you can do this with `GET` or `POST`. If you've used `GET` as a method, you can write the parameters directly into the URL. In the following code snippet, a few parameters are passed to a script:

```
...
  xmlhttp.open("GET","test.php?name=wolfe&zip=97217",true);
  xmlhttp.send();
...
```

By using `POST`, on the other hand, you can specify the data in the `send()` method of the `XMLHttpRequest` object. In addition, you must send a special HTTP header with `POST`. You can do this via the `setRequestHeader()` method. The required HTTP header is `Content type`, and the corresponding value is `"application/x-www-form-urlencoded"`. This multipurpose internet mail extension (MIME) type is used for form data. For JSON, the MIME type would again look different. So, here's what you can do with `POST` to pass the data to the script:

```
...
  xmlhttp.open("POST","test.php", true);
  xmlhttp.setRequestHeader("Content type",
      "application/x-www-form-urlencoded");
  xmlhttp.send("name=wolfe&zip=97217");
...
```

20.1.5 Determining the Status of the “XMLHttpRequest” Object

Now that you know how to make a connection and request to the server, you're still missing an important component without which the asynchronous data transfer wouldn't work properly. It's a callback function that's still missing, which gets called when results come back from the web server. You pass the callback function that is called in the process to the `XMLHttpRequest` property `onreadystatechange`. In this example, you use an anonymous function for this purpose:

```
xmlhttp.onreadystatechange = function() { ... };
```

You can also pass a function name as a reference as follows, if required:

```
function aFunction() { ... }
...
xmlhttp.onreadystatechange = aFunction;
```

It should be noted here that the `readystatechange` event gets triggered whenever the state of the `XMLHttpRequest` object changes.

In this callback function, you first check the state of the `XMLHttpRequest` object via the `readyState` property. There are five different states, as listed in Table 20.1.

Value	Status	Description
0	`UNSENT`	The `open()` function hasn't been called yet.
1	`OPENED`	The `send()` function hasn't been called yet.
2	`HEADERS_RECEIVED`	The `send()` function has already been called, and the headers and status are available.
3	`LOADING`	The download is in progress, but the `responseText` isn't yet complete.
4	`DONE`	The process has been fully completed.

Table 20.1 You Can Determine the Status of a Request via the "readyState" Attribute

In addition to the status of the `XMLHttpRequest` object, the status of the response to the request is significant; for example, the value `200` will be returned if the request was successful. For example, the classic value `404` will be returned if the requested page couldn't be found.

Status Code	Message	Meaning
`200`	`Ok`	The request was successfully processed and the result of the response was transmitted.
`400`	`Bad Request`	The request message was incorrect.
`403`	`Forbidden`	The request could not be performed because no authorization for it exists.
`404`	`Not Found`	The requested resource wasn't found on the web server.

Table 20.2 List of Common HTTP Status Codes for "status"

Here's the snippet that is used to check the state of the `XMLHttpRequest` object:

```
...
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
      document.querySelector('#refreshtime').innerHTML =
            xmlhttp.responseText;
    }
  }
...
```

Listing 20.7 /examples/chapter020/20_1/js/getServerTime.js

You use the `onreadystatechange` property to pass the callback function to be called when the server response is available. This function is called whenever the state of the `readyState` property has changed. In this example, you insert the text in `responseText` of the `XMLHttpRequest` object in the HTML document at the HTML element with ID `dynamic01` if `readyState` is `4` and the status of the response in `status` is `200`. The `responseText` attribute contains the web server's response to the request as text or `null` if the request was unsuccessful.

Other Properties of the "XMLHttpRequest" Object

In addition to the `onreadystatechange`, `readyState`, `status`, and `responseText` properties presented here, there are a number of other attributes the `XMLHttpRequest` object brings with it, which won't be described any further here. For more information, you should visit *https://developer.mozilla.org/de/docs/DOM/XMLHttpRequest*.

20.1.6 Processing the Response from the Server

The response from the server can be found in the `responseText` or `responseXML` property of the `XMLHttpRequest` object. The data contained therein can then be processed further. You can use the following lines to insert the returned text as new text in the HTML element. The ID is `dynamic01`:

```
...
document.querySelector('#refreshtime').innerHTML =
    xmlhttp.responseText;
...
```

Listing 20.8 /examples/chapter020/20_1/js/getServerTime.js

If the return from the server isn't XML, you should always use the `responseText` property, where the returned data is an ordinary string. If, on the other hand, the return from the server is XML-encoded and you want to parse this data as an XML object, you should use the `responseXML` property instead.

20.1.7 The Ajax Example during Execution

Based on the preceding descriptions, you should have understood the example printed at the beginning: */examples/chapter020/20_1/js/getServerTime.js*. For this example, you're still missing the PHP script *server-time.php*, which was kept quite short and basically only returns an indication of the date and time of the web server:

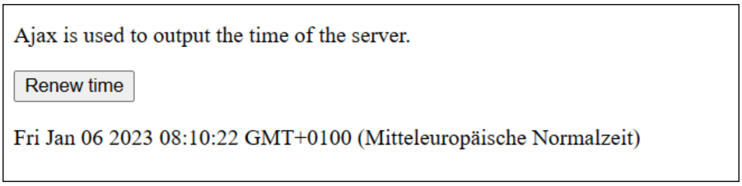
```
<?php
  echo date('l jS \of F Y h:i:s A');
?>
```

Listing 20.9 /examples/chapter020/20_1/php/server-time.php

When you run the example, you'll find the view shown in Figure 20.4 in your web browser. The first paragraph with the text `With Ajax, the time ...` is supposed to change using the `XMLHttpRequest` object when the user clicks the button. The timestamp after the button has already been set using the JavaScript function `timestamp()` when the web page gets loaded in the `body` element via `onload`.

In Figure 20.5, the button was clicked and the time of the web server was displayed above it. For the entire process, Ajax was used along with the `XMLHttpRequest` object. You can click the button as many times as you want, and every time the time output above the button will be updated accordingly.

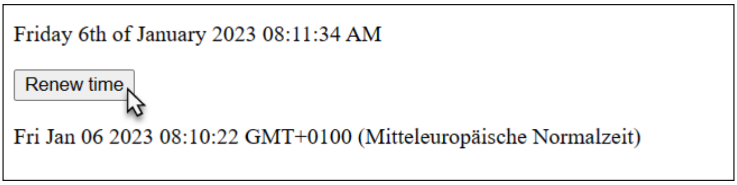
In this case, however, the website never gets completely reloaded, but only individual information or components of the website are updated. In the example, therefore, only the web server time above the button gets updated all the time. A complete reload of the page would also update the time below the button. The following two figures shows this process.



Ajax is used to output the time of the server.

Fri Jan 06 2023 08:10:22 GMT+0100 (Mitteleuropäische Normalzeit)

Figure 20.4 The Web Page Was Loaded



Friday 6th of January 2023 08:11:34 AM

Fri Jan 06 2023 08:10:22 GMT+0100 (Mitteleuropäische Normalzeit)

Figure 20.5 Our First Ajax Application during Execution

20.1.8 A More Complex Ajax Example with XML and DOM

Now you'll create a slightly more complex example using XML and DOM. You'll also use Ajax to create a more interactive application. In the following example, you'll see how a web page can communicate with the web server while the user is typing something into an input field via the keyboard. By the way, Google does this similarly by providing suggestions while you're typing something into the search box, but that's much more complex, of course.

For this example, the following input field is created:

```
...
<h1>Unit conversion</h1>
<form>
<fieldset>
<legend>Convert meters to miles and yards</legend>.
  <label>Meters</label>
  <input type="number" id="meters" placeholder="Value in meters"
         onkeyup="recalculate(this.value);"> m
  <br><label>Miles</label>
  <input type="number" id="miles"
         placeholder="conversion to miles" readonly >mi
  <br><label>Yards</label>
  <input type="number" id="yards"
         placeholder="Conversion to yards" readonly> yds
</fieldset>
</form>
...
```

Listing 20.10 /examples/chapter020/20_1_8/index.html

In this user input, users can make a numerical input in meters. After each keystroke (= `onkeyup`), you send the value with the event handler `recalculate()` via Ajax to the web server.

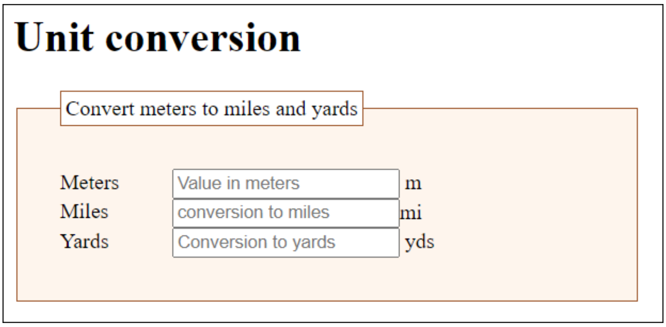

Unit conversion
Convert meters to miles and yards
Meters
Value in meters
m
Miles
conversion to miles
mi
Yards
Conversion to yards
yds

Figure 20.6 Users Can Enter a Numerical Value in Meters

Here's the code for `recalculate()`, which takes care of the remaining tasks:

```
let xmlhttp = null;

function recalculate(str) {
    if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
    if (xmlhttp === null) {
        console.log("Error creating an XMLHttpRequest object");
    }
    xmlhttp.open("GET", "php/calculate.php?meters=" + str, true);
    xmlhttp.onreadystatechange = parseRecalculate;
    xmlhttp.send();
}

function parseRecalculate() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        let xml = xmlhttp.responseXML;
        let miles_response = xml.querySelector('miles');
        let yards_response = xml.querySelector('yards');

        document.querySelector('#miles').value =
           miles_response.firstChild.nodeValue;
        document.querySelector('#yard').value = yard_
response.firstChild.nodeValue;
    } else {
        document.querySelector('#miles').value = 0;
        document.querySelector('#yards').value = 0;
    }
}
```

Listing 20.11 /examples/chapter020/20_1_8/js/convert.js

There isn't really much new in `recalculate(str)` itself. First, you create a new `XMLHttpRequest` object. Then you asynchronously transfer the entered value to the web server via Ajax after each keystroke. This request was still made with the `GET` method because it's only a value. You assemble the entered value in `str` directly after the URL of the `open()` method. For example, if the user has entered `1000`, the URL looks as follows:

```
calculate.php?meters=1000
```

Instead of an anonymous function, this example assigned a reference to an existing function as a callback function named `onreadystatechange` with `parseRecalculate`. This function will take care of everything else once the request has been sent using `send()` and the response from the web server is available. Here's the PHP script *calculate.php* for that, which calculates the passed value in `meters` on the web server and responds with an XML-coded calculation:

```
<?php
  header("Content type: text/xml");
  $meters = $_REQUEST['meters'];
  $miles = $meters * 0.000621371192237339;
  $yards = $meters * 1.0936133;
  echo "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
?>

<conversion>
    <meters><?php echo $meters; ?></meters>
    <miles><?php echo $miles; ?></miles>
    <yards><?php echo $yards; ?></yards>
</conversion>
```

Listing 20.12 /examples/chapter020/20_1_8/php/calculate.php

You can retrieve the passed value using `$_REQUEST['meters']` and pass it to the `$meters` variable. Then you convert the value into miles and yards and store these two values in the `$miles` and `$yards` variables, respectively. After that, you create the return XML-encoded document as a response. For example, if you've entered the value `1000` for meters, the following XML-encoded output will be generated in response:

```
<?xml version="1.0" encoding="UTF-8"?>
<conversion>
    <meters>1000</meters>
    <miles>0.62137119223733</miles>
    <yards>1093.6133</yards>
</conversion>
```

You then evaluate this returned XML document using the `parseRecalculate()` callback function you had set up using `onreadystatechange`. Here's the section with the `parseRecalculate()` function:

```
...
function parseRecalculate() {
  if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
   // Response from the server
    let xml = xmlhttp.responseXML;
    let miles_response =
       xml.querySelector('miles');
    let yards_response =
        xml.querySelector('yards');
    // Write results
    document.querySelector('#miles').value =
        miles_response.firstChild.nodeValue;
    document.querySelector('#yards').value =
        yards_response.firstChild.nodeValue;
  }
  else { // In case of an error
    document.querySelector('#miles').value = 0;
    document.querySelector('#yards').value = 0;
  }
}
...
```

Listing 20.13 /examples/chapter020/20_1_8/js/convert.js

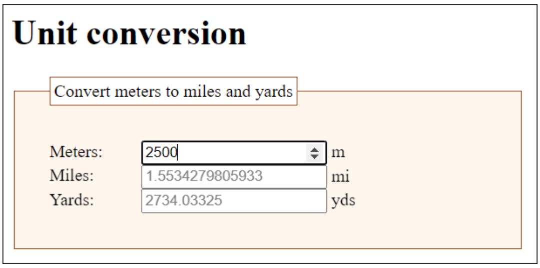
This function has a relatively logical structure as well. First, you use `readyState` and `status` to check if the response from the server is ready. If so, you'll find the response in `responseXML` of the `XMLHttpRequest` object because the response is now XML-encoded. Because you're using an XML file here, you can immediately evaluate the response using the Document Object Model (DOM). In this case, you want to get the value of the node with the tag `miles` and the other node with the tag `yards`. You can read these values via DOM as follows:

```
let miles_response=xml.querySelector('miles');
let yards_response=xml.querySelector('yards');
```

Here, you can see that you can access the DOM methods directly by using `responseXML`. Finally, you write the values in `miles_response` and `yards_response` that have been read from the XML document via DOM into the appropriate form fields as follows:

```
document.querySelector('#miles').value =
    miles_response.firstChild.nodeValue;
document.querySelector('#yards').value =
    yards_response.firstChild.nodeValue;
```

Admittedly, this example is considerably more complex because in addition to JavaScript and server-side script programming, XML and DOM have been added. Nevertheless, the example gives you a first impression of what can be done with Ajax and that it can be worthwhile to deal with it more comprehensively.



Unit conversion

Convert meters to miles and yards

Meters: 2500 m
Miles: 1.5534279805933 mi
Yards: 2734.03325 yds

Figure 20.7 The Ajax Application during Execution

20.1.9 The JSON Data Format with Ajax

In the previous example, you performed a data exchange using XML. At the beginning, I mentioned that Ajax also has other data formats for exchanging information between server and client. A very popular and simpler alternative to XML is JSON, which is probably now more widely used than XML. For this reason, here's a short introduction to this data format.

JSON enables you to specify objects and arrays as ordinary strings, just as you know it from JavaScript. The process is referred to as *serialization*.

You can specify an array between square brackets:

```
["text1", "text2", "text3", "text4"]
```

Objects are placed in curly brackets:

```
{"property1" : "value", "property2" : "value" }
```

Even though JSON uses a JavaScript syntax, like XML, this data format is independent of the language. The data format can be read by any programming language.

Here's a JSON example in which you create a `directory` object with three entries of zip codes from a combination of city-zip code pairs:

```
{"directory":[
    {"city":"Portland", "zipCode":97217},
    {"city":"San Francisco", "zipCode":94104},
    {"city":"Philadelphia", "zipCode":19099}
]}
```

By the way, JSON requires that the property names (e.g., `"city"` and `"zipCode"`) be enclosed in double quotes. In addition to objects (in curly brackets) and arrays (in square brackets), you can use a number (integer and floating point), a string (in double quotes), a Boolean value (`true` or `false`), and the null value (`null`) as data types.

Because JSON uses JavaScript syntax, it's easy to create such an array of objects. In JavaScript, all you need to do is the following:

```
let directory =  [
    {"city":"Portland", "zipCode":97217},
    {"city":"San Francisco", "zipCode":94104},
    {"city":"Philadelphia", "zipCode":19099}
];
```

Here's a simple example that demonstrates how you can access the individual entries using JavaScript:

```
let directory = [
    { "city": "Portland", "zipCode": 97217 },
    { "city": "San Francisco", "zipCode": 94104 },
    { "city": "Philadelphia", "zipCode": 19099 }
];

document.querySelector('#output').innerHTML = "<ul>" +
    "<li>" + directory[0].city + " = " + directory[0].zipCode + "</li>" +
    "<li>" + directory[1].city + " = " + directory[1].zipCode + "</li>" +
    "<li>" + directory[2].city + " = " + directory[2].zipCode +
    "</li></ul>";
```

Listing 20.14 /examples/chapter020/20_1_9/js/plz.js

Let's look at an access such as the following:

```
directory[0].city + " = " + directory[0].zipCode
```

This returns the following in this example:

```
Portland = 97217
```

Similarly, you can also modify the data as follows:

```
directory[0].city = "Houston"; // Portland becomes Houston
```

Usually, you use a `for` loop for traversing each element in the JSON data format, which might look like the following:

```
...
let txt = "<ul>";
for (let i = 0; i<directory.length; i++) {
  txt += "<li>" + directory[i].city + " = "
      + directory[i].zipCode + "</li>";
}
txt += "</ul>";
document.querySelector('#output').innerHTML = txt;
...
```

In practice, you'll use the JSON data format pretty often to read data from a web server to display it on a web page. This returns us to Ajax, which is what this chapter is about. The following *data.json* file is located on the web server and is supposed to be read and output on the web page:

```
[
  {
    "url":"http://www.portland.com/",
    "city":"Portland",
    "zipCode":97217
  },
  {
    "url":"http://www.sanfrancisco.com/",
    "city":"San Francisco",
    "zipCode":94104
  },
  {
    "url":"http://www.philadelphia.com/",
    "city":"Philadelphia",
    "zipCode":19099
  }
]
```

Listing 20.15 /examples/chapter020/20_1_9/json/data.json

You're probably wondering how to convert this text (i.e., the JSON string), which, if everything goes right, is contained in `responseText` of the `XMLHttpRequest` object, into a JavaScript object. For this purpose, JavaScript provides the `JSON.parse()` method:

```
let obj = JSON.parse(text);
```

You can use the `JSON.parse()` method to convert a JSON text into a JavaScript object.

Other Ways to Parse JSON

For older web browsers that don't support `JSON.parse()`, you could use the `eval()` function to convert a JSON string into a JavaScript object:

```
let obj = eval ("(" + text + ")");
```

However, the use of `eval()` isn't without problems because it can be used to execute any JavaScript code. If the data comes from your own web server, this can still be okay, but you should never use data from an external URL with it. It's therefore safer to use one of the many JavaScript frameworks that have their own JSON parser integrated.

Here is the Ajax example where you read this file from the web server using a JSON string, parse it, and output it to a web page:

```
...
<h1>JSON example with Ajax</h1>
<p id="output"></p>
<script src="js/plzJSON.js"></script>

...
```

Listing 20.16 /examples/chapter020/20_1_9/index2.html

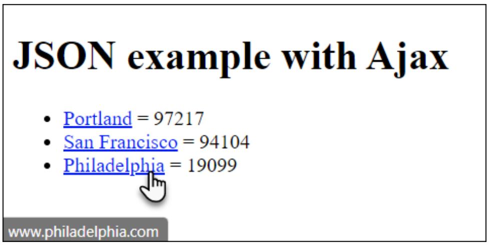
```
let xmlhttp = new XMLHttpRequest();
let url = "json/data.json";

xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        let myArr = JSON.parse(xmlhttp.responseText);
        makeOutput(myArr);
    }
}
xmlhttp.open("GET", url, true);
xmlhttp.send();

function makeOutput(arr) {
    let out = '<ul>';
    for (let i = 0; i < arr.length; i++) {
        out += '<li>' + '<a href="' + arr[i].url + '">' +
            arr[i].city + '</a>' + " = " + arr[i].zipCode + '</li>';
    }
    out += '</ul>';
    document.querySelector('#output').innerHTML = out;
}
```

Listing 20.17 /examples/chapter020/20_1_9/js/plzJSON.js

Here, you connect to the server and make an HTTP request for the *data.json* file. If the request was successful, you can parse the string into JSON and create an object:

```
let myArr = JSON.parse(xmlhttp.responseText);
```

You pass this data to the `myOutput()` function, where the content for the web page gets compiled and displayed in HTML. You can see the example during execution in Figure 20.8.



JSON example with Ajax

- Portland = 97217
- San Francisco = 94104
- Philadelphia = 19099

Figure 20.8 The Content of the JSON File data.json Was Read, Parsed, and Displayed on the Web Page Using Ajax

This example is kept simple to give you some understanding of how to use the JSON and Ajax data formats.

Fetch API

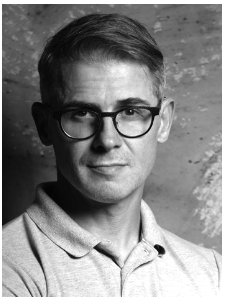
By the way, in practice, you don't have to take the complex way via the `XMLHttpRequest` object. For Ajax requests, for example, the Fetch application programming interface (API) at *https://fetch.spec.whatwg.org* is a good choice. It's much easier to use and simplifies sending Ajax-based requests.

20.2 Summary

In this chapter, you got to know Ajax, a technology that enables you to transfer data asynchronously between the web browser and the web server. You now know how to make HTTP requests while an HTML web page is displayed in order to modify that page without having to reload it.

Having read this chapter, you'll know the basic principles of implementing and using Ajax in practice.

The Author



Jürgen Wolf is a web and software developer and the author of several seminal works about programming and photography.

Index

::first-letter (pseudo-element) 330
::first-line (pseudo-element) 330
::selection (pseudo-element) 330
:active (pseudo-class) 319, 588
:any-link (pseudo-class) 318
:blank (pseudo-class) 324
:empty (pseudo-class) 323
:first-child (pseudo-class) 324
:first-of-type (pseudo-class) 326
:focus (pseudo-class) 319
:hover (pseudo-class) 319, 320, 588
:invalid (pseudo-class) 255, 261
:lang() (pseudo-class) 329
:last-child (pseudo-class) 324
:last-of-type (pseudo-class) 326
:link (pseudo-class) 318
:matches() (pseudo-class) 329
:not() (pseudo-class) 329
:nth-child() (pseudo-class) 324
:nth-last-child() (pseudo-class) 324
:nth-last-of-type() (pseudo-class) 326
:nth-of-type() (pseudo-class) 326
:only-of-type (pseudo-class) 326
:placeholder-shown (pseudo-class) 319
:required (pseudo-class) 262
:root (pseudo-class) 323
:target (pseudo-class) 321
:valid (pseudo-class) 255, 261
:visited (pseudo-class) 318
!important 355
@font-face 526
@import (CSS rule) 293, 462, 598
@media (CSS rule) 295, 460, 462
@supports() 441, 596
@viewport 468
&gt; 161
&lt; 161
&nbsp; 115, 166
&quot; 161
&shy; 115
#anchorname 188
<!-- string 65
<noscript> 645
, list-style-type 558
<th>email address</th> 183, 255
62.5% trick 543

A

a (tag) 177
  #anchorname 188
  download 186, 192
  href 178, 192
  href=mailto 183
  hreflang 192
  media 192
  phone number 191
  rel 193
  Skype 191
  target 182, 193
  title 186
  type 186, 193
abbr (tag) 140
  title 140
accesskey 265
addEventListener() 738
address (tag) 110
Adobe Brackets 45
Ajax 779
  callback function 785
  determining the status 785
  DOM 788
  example 781
  HTTP request 783
  HTTP response 787
  JSON 793
  onreadystatechange 785, 790
  open() 784
  readyState 785
  responseText 787
  responseXML 787, 792
  send() 784
  status 787
  XMLHttpRequest object 783
align-content 448
align-items 511
align-self 449, 511
all 353
Anchor 188
and (media query) 463
Angular dimensions 372
any-hover 465
any-pointer 465
appendChild() 755, 760
Application server 41

area (tag) ... 205
  alt ... 209
  coords ... 207, 209
  download ... 209
  href ... 208, 209
  hreflang ... 209
  media ... 209
  rel ... 209
  shape ... 207, 209
  target ... 209
  type ... 209
Array ... 677
Array literal notation ... 689
article (tag) ... 98, 135
ASCII encoding ... 158
aside (tag) ... 99, 135
aspect-ratio ... 464
audio (tag) ... 229
  autoplay ... 230
  controls ... 230
  loop ... 230
  muted ... 231
  preload ... 231
  src ... 231
  type ... 230, 231
Author stylesheet ... 355
Autocompletion ... 258
Automatic redirection ... 90

B

b (tag) ... 146
background ... 402
  background-attachment ... 398, 401
  background-color ... 397, 398
  background-image ... 398, 498
  background-position ... 398, 401
  background-repeat ... 398, 400
  background-size ... 403, 498
  linear-gradient() ... 406
  radial-gradient() ... 408
  repeating-linear-gradient() ... 407
  repeating-radial-gradient() ... 408
Background color ... 397
Background graphic ... 399
  fixing ... 401
  positioning ... 401
  tiling ... 399
Background image ... 397
base (tag) ... 76
  href ... 76, 77
  target ... 77
bdi (tag) ... 144
bdo (tag) ... 144
  dir ... 144
Blisk ... 598
blockquote (tag) ... 117
  cite ... 117
Blog ... 33
body (tag) ... 68, 96
Boolean ... 709
Boolean data type, JavaScript ... 660
Border ... 395
  decorative border ... 396
border ... 379
border (tag) ... 393
border-bottom ... 379
border-box ... 472
border-collapse ... 568
border-color feature ... 394
border-image (tag) ... 396
border-image-slice (tag) ... 396
border-image-source (tag) ... 396
border-image-width (tag) ... 396
border-left ... 379
border-radius ... 411
  border-bottom-left-radius ... 411
  border-top-left-radius ... 411
  border-top-right-radius ... 411
border-right ... 379
border-spacing ... 569
border-style feature ... 394
border-top ... 379
border-width feature ... 394
Box model ... 375
  alternate ... 386
  box-sizing:border-box ... 386
  classic ... 376
box-shadow ... 409
box-sizing ... 386, 472
  border-box ... 388
  content-box ... 388
br (tag) ... 113
break, JavaScript ... 675
Breakpoint ... 471
Browser stylesheet ... 355
Bullet point ... 122
button (tag) ... 247

C

calc() ... 516, 541
Camel case ... 734
canvas (tag) ... 221

Capital letters ... 544
caption (tag) ... 175
caption-side ... 571
Cascade ... 291
Cascading ... 345, 354
Cascading Style Sheets → CSS
Centimeters ... 364
Central stylesheet ... 598
Character encoding ... 86, 157
Character entity ... 160
charset ... 159
Chrome ... 597
circle (SVG) ... 218
cite, q (tag) ... 147
cite (tag) ... 140
  title ... 140
Class ... 704
  class selector ... 302
clear ... 487
  both ... 439
  left ... 439
  none ... 439
  right ... 439
cloneNode() ... 764
code (tag) ... 141
col (tag) ... 172
colgroup (tag) ... 172
Collapsing margins ... 381
Color ... 366
  hexadecimal notation ... 367
  HSL mixture ... 370
  named ... 366
  selecting ... 251
  selection dialog ... 251
color ... 465
Color detail ... 366
  named colors ... 366
  RGB mixture ... 368
  transparency ... 369
color-index ... 465
ColorZilla ... 370
column-count ... 555
column-gap ... 555
columns ... 556
column-width ... 556
Combinator ... 298, 332
Comment ... 65
  CSS ... 284
Comparison operators, JavaScript ... 667
Conditional statements, JavaScript ... 665
Console, JavaScript ... 647
console object ... 647
const ... 655
Content area ... 376
continue, JavaScript ... 675
Corporate website ... 32
createElement() ... 755
createTextNode() ... 755
CSS ... 42, 279
  alternate stylesheet ... 291
  cascade ... 291
  cascading ... 354
  code formatting ... 285
  commenting code ... 284
  compression ... 600
  inheritance ... 345
  integrating in HTML ... 285
  in web browser ... 295
  level 1 (CSS 1) ... 280
  level 2 (CSS 2) ... 280
  level 3 (CSS3) ... 280
  manipulating ... 733
  media query ... 295
  rule ... 282, 283, 289
  selectors ... 283, 297
  style attribute ... 286
CSS feature
  !important ... 355
  angular dimensions ... 372
  background ... 397
  background-repeat ... 400
  bottom ... 420
  color details ... 366
  content ... 330
  default value ... 349
  forcing inheritance ... 350
  height ... 376
  inheritance ... 345
  keyword (value) ... 365
  left ... 420
  margin ... 379
  named colors ... 366
  opacity ... 405
  order ... 454
  position ... 417
  restoring the default value ... 352
  right ... 420
  short notation ... 373
  string (value) ... 365
  top ... 420
  unit of measurement ... 363
  width ... 376
  z-index ... 431

CSS preprocessor ... 605, 606
  installing ... 607
  online ... 607
CSS pseudo-class ... 318
  :active ... 319
  :any-link ... 318
  :blank ... 324
  :checked ... 320
  :disabled ... 320
  :empty ... 323
  :enabled ... 320
  :first-child ... 324
  :first-of-type ... 326
  :focus ... 319
  :hover ... 319, 567
  :invalid ... 255, 256, 261
  :lang() ... 329
  :last-child ... 324
  :last-of-type ... 326
  :link ... 318
  :matches() ... 329
  :not() ... 329
  :nth-child() ... 324
  :nth-last-child() ... 324
  :nth-last-of-type() ... 326
  :nth-of-type() ... 326
  :only-child ... 324, 326
  :placeholder-shown ... 319
  :required ... 262
  :root ... 323
  :target ... 321
  :valid ... 255, 256, 261
  :visited ... 318
CSS pseudo-element ... 330
  : ... 330
CSS reset ... 600
  normalization ... 602
CSS web browser test ... 593

D

Database ... 41
datalist (tag) ... 259
Data type, JavaScript ... 657
DCOMContentLoaded ... 741
dd (tag) ... 128
Declaration ... 283
  components ... 284
Decrement operator, JavaScript ... 672
default, track (tag) ... 226
Degree ... 372
del (tag) ... 150
Designing a border ... 393
details (tag) ... 175, 275
  open ... 275
dfn (tag) ... 143
dialog (tag) ... 276
Directory name ... 73
Directory structure ... 73
display ... 495, 513
  block ... 513, 585
  flex; ... 443
  grid ... 501
  inline ... 513
  inline-block ... 513, 585
  none ... 515
  none (CSS element) ... 459
div (tag) ... 118, 132, 583
dl (tag) ... 128
doctype (tag) ... 66
document.body ... 755
document.documentElement ... 755
document object ... 717
Document Object Model →DOM
Document outline ... 105, 107
DOM ... 715
  Ajax ... 788
  document object ... 717
DOM functions, setAttribute() ... 730
DOM inspector ... 58, 61, 716
DOM manipulation ... 715, 754
DOM method ... 718
DOM object collection ... 727
  baseURI ... 729
  body ... 729
  cookie ... 729
  doctype ... 729
  documentElement ... 729
  documentURI ... 729
  domain ... 729
  domConfig ... 729
  embeds ... 729
  forms ... 729
  head ... 729
  images ... 729
  implementation ... 729
  inputEncoding ... 729
  lastModified ... 729
  links ... 729
  readyState ... 729
  referrer ... 729
  scripts ... 730
  title ... 730
  URL ... 730

DOM property ... 718, 756
  childNodes ... 756
  firstChild ... 792
  nodeValue ... 792
DOM tree ... 58, 716
do-while loop, JavaScript ... 674
Download link ... 186
dt (tag) ... 128
Dynamic website ... 39

E

ECMAScript ... 639
E-commerce website ... 35
ellipse (SVG) ... 218
em (tag) ... 145
embed (tag) ... 228, 232
empty-cells ... 570
em quad ... 364
Event ... 735
Event handler ... 736
  addEventListener() ... 738
Event object ... 746
  altKey ... 747
  bubbles ... 747
  button ... 747
  cancelable ... 747
  clientX ... 747
  clientY ... 747
  ctrlKey ... 747
  currentTarget ... 747
  keyCode ... 748
  metaKey ... 748
  preventDefault() ... 748
  screenX ... 748
  screenY ... 748
  shiftKey ... 747
  target ... 748
  type ... 748
Event propagation ... 749

F

false, JavaScript ... 660
Favicon ... 213
favicon.ico ... 214
Feature query ... 596
fieldset (tag) ... 266, 583, 588
figcaption (tag) ... 121, 176, 202
figure (tag) ... 121, 176, 202
File name ... 72
Firefox ... 597
firstChild ... 756
flex ... 451
flex-basis ... 451
Flexbox ... 443, 562
  order ... 454
flex-direction
  column ... 444
  column-reverse ... 444
  row ... 443
  row-reverse ... 444
flex-flow ... 446
flex-grow ... 451
flex-shrink ... 451
flex-wrap ... 445
float ... 434
  inherit ... 436
  left ... 435
  none ... 436
  right ... 435
flow-root, display ... 440
font ... 546
Font Awesome ... 533
Font class ... 522
font-family ... 522
  web fonts ... 527
Font formatting ... 546
Fonts ... 522
  royalty-free ... 532
  web font ... 526
Font size ... 536
  em ... 537
  keyword ... 537
  pixels ... 540
  points ... 540
  relative (em) ... 537
  rem ... 539
font-size ... 536
Font stack ... 522
font-stretch ... 548
Font style
  bold ... 543
  italic ... 543
font-style ... 543
font-variant ... 544
font-weight ... 543
footer (tag) ... 108, 135
for loop, JavaScript ... 673
Form ... 237
  autocomplete ... 258
  button ... 247
  checkbox ... 245
  color selection dialog ... 251

Form (Cont.)
- *date input field* 252
- *defining a space* 238
- *disabling elements* 264
- *dropdown list* 242
- *email input field* 255
- *entering date and time* 253
- *error during input* 261
- *file upload* 246
- *grouping element* 266
- *hidden input field* 248
- *input fields* 250
- *keyboard shortcut* 265
- *mailer* 269
- *month input field* 254
- *multiline text input field* 241
- *multiple submit buttons* 249
- *number input field* 256
- *password input field* 240
- *phone number input field* 256
- *PHP* 268
- *radio buttons* 244
- *read only* 265
- *regular expressions* 260
- *search input field* 254
- *selection list* 242
- *setting the input focus* 258
- *slider* 256
- *tab sequence* 265
- *text input field* 240
- *text label* 245
- *time input field* 252
- *URL input field* 255
- *using placeholders* 260
- *week input field* 254

form (tag) 238, 248
- *accept-charset* 239
- *action* 238, 269, 271
- *enctype* 239
- *id* 248
- *method* 238
- *method=* 271
- *target* 239

Forms, JavaScript 770
fr 502
Function 677

G

g (SVG) 219
GDPR consent 272
General Data Protection Regulation (GDPR) 532
getAttribute() 765
getElementById() 719
getElementsByClassName() 723
getElementsByName() 724
getElementsByTagName() 720
GET method 271
Google Fonts 529
Gradian 372
Gradient 406
Graphic
- *embedding* 196
- *link-sensitive* 204

grid 501
grid (layout) 501
grid-area 507
grid-column 506
grid-column-end 504
grid-column-gap 511
grid-column-start 504
grid-gap 511
grid-row 506
grid-row-end 504
grid-row-gap 511
grid-row-start 504
grid-template-columns 502
grid-template-rows 502
Grouping columns 172

H

h1 (tag) 104
h2 (tag) 104
h3 (tag) 104
h4 (tag) 104
h5 (tag) 104
h6 (tag) 104
hasAttribute() 766
hasChildNodes 757
head (tag) 67, 69
header (tag) 108, 135
Heading 104
Height 378
height 464, 571
hidden, main (tag) 121
hover 465
hr (tag) 116
href, area (tag) 208
hsl() 370
hsla() 370

HTML ... 31
- *adding CSS* ... 285
- *head data* ... 67
- *input fields* ... 239
- *markup language* ... 41
- *page elements* ... 57
- *validating* ... 49

html (tag) ... 67
HTML5 web browser test ... 594
HTML attribute ... 64
- *datetime* ... 151
- *height* ... 215
- *id* ... 221
- *manipulating* ... 732, 765
- *meta element* ... 92
- *table elements* ... 169
- *type* ... 232
- *width* ... 215

HTML document ... 638
- *body* ... 96
- *framework* ... 65
- *in browser* ... 56
- *structure* ... 55

HTML element ... 59
- *<iframe>* ... 233
- *<source>* ... 229
- *head* ... 69
- *incorrect nesting* ... 61
- *interactive* ... 275
- *nesting* ... 60
- *omitting tag* ... 62
- *structuring pages* ... 95
- *structuring text* ... 111

HTML form ... 237
HTML tag ... 59
HTTP request ... 270
- *method* ... 239

Hyperlink ... 163, 177
Hyphen &shy; ... 115

I

i (tag) ... 146
Icon ... 213, 532
Icon font ... 532
ID selector ... 305
if branch, JavaScript ... 668
iframe (tag) ... 228
- *sandbox* ... 234
- *seamless* ... 235

Image
- *embedding* ... 196

Image (Cont.)
- *hiding* ... 495
- *labeling* ... 202
- *responsive* ... 489
- *scaling* ... 201
- *scaling with CSS* ... 571

Image map ... 204
img (tag) ... 196
- *alt* ... 196, 197, 204
- *height* ... 200, 204
- *hiding* ... 495
- *ismap* ... 204
- *making responsive* ... 489
- *name* ... 204
- *scaling with CSS* ... 571
- *src* ... 196, 204
- *src (SVG)* ... 215
- *SVG* ... 215
- *title* ... 198
- *usemap* ... 204
- *width* ... 200, 201, 204

Inches ... 364
Increment operator, JavaScript ... 672
inherit ... 350
Inheritance ... 345
- *CSS* ... 345
- *inherit* ... 350

initial ... 352
initial-scale ... 468
Inline style ... 286
innerHTML ... 730
input (tag)
- *accept* ... 246
- *accesskey* ... 265
- *autocomplete* ... 258
- *autofocus* ... 258
- *checked* ... 244
- *disabled* ... 264
- *enctype* ... 246
- *for* ... 245
- *formaction* ... 249
- *formmethod* ... 249
- *formnovalid* ... 250
- *formtarget* ... 250
- *input type=* ... 240, 247
- *JavaScript* ... 771, 773
- *list* ... 259
- *max* ... 259
- *maxlength* ... 240
- *min* ... 259
- *multiple* ... 255, 260
- *name* ... 240, 244, 246, 248

input (tag) (Cont.)
  novalidate 263
  pattern 256, 260
  placeholder 260
  readonly 265
  required 255, 261, 262
  size 240
  step 259
  tabindex 265
  type= 246, 248, 251–256
  type=checkbox 245
  type=radio 244
  value 240, 244, 246, 248, 251, 771
ins (tag) 150
insertBefore() 760
ISO-8859-1 158

J

JavaScript 43, 637, 638
  -- 672
  ++ 672
  <noscript> 645
  === operator 710
  alert() 646
  arguments object 681
  arithmetic operator 663
  array 689
  array (multidimensional) 691
  arrow functions 686
  bool 660
  Boolean 709
  break 675
  canvas (tag) 221
  class 704
  class syntax 704
  comments 652
  comparison operators 667
  conditional statement 665
  confirm() 646
  console 647
  console.log() 647
  const 655
  constant 655
  constructor function 703
  continue 675
  converting data types 662
  customizing the load behavior 645
  data type 657
  Date 711
  Date object 691
  decrement operator 672

JavaScript (Cont.)
  default parameter 681
  defining functions 678
  disabled 83
  do-while loop 674
  false 660, 666
  for ... in (loop) 693
  for ... of (loop) 693
  for loop 673
  forms 770
  Function 711
  function 677, 678
  function expression 679
  function parameter 680
  get 706
  getter 705
  if branch 668
  increment operator 672
  indexOf() 699
  input dialog 646
  integration in HTML 641
  isNaN() 663
  length 700
  let 653, 683
  logical operator 669
  loop 672
  Map object 712
  Math 711
  Math object 665
  multiple branching 670
  null 661
  number 709
  numbers 657
  object 702, 709
  object-oriented programming 702
  output 646
  parseFloat() 663
  parseInt() 663
  pop() 694
  position 644
  prompt() 646
  push() 694
  queue 696
  regular expression 701
  rest parameter 682
  return statement 683
  return value (function) 683
  scope (variables) 683
  selection operator 669
  set 706
  Set object 712
  setter 705

JavaScript (Cont.)
  shift() 695
  sort() 697
  splice() 696
  stack 695
  standard dialog 646
  strict mode 656
  string 658, 700, 709
  substr() 701
  substring() 701
  switch 670
  this 703, 708
  traversing arrays 693
  true 660, 666
  typeof 657
  undefined 661
  unshift() 695
  use strict 656
  var 653, 683
  variables 652
  while loop 674
  within HTML 643
JavaScript engine 640
JavaScript event 735
  blur 745
  bubbling phase 749
  capturing phase 749
  change 745
  click 742
  dblclick 742
  default action 748
  DOMContentLoaded 741
  error 740
  event handler 736
  event propagation 749
  focus 745
  keydown 744
  keypress 745
  keyup 745
  load 740
  mousedown 742
  mousemove 742
  mouseout 742
  mouseover 742
  mouseup 742
  onclick 773
  onkeyup 789
  onload 741, 783
  onunload 741
  preventing the default action 748
  properties 745
  reset 745
JavaScript event (Cont.)
  resize 740
  scroll 741
  select 745
  submit 745
  touchcancel 744
  touchend 744
  touchmove 744
  touchstart 744
  unload 741
JavaScript objects
  Date 711
  Function 711
  Map 712
  Math 711
  Set 712
JSON 793
  JSON.parse() 795
Jump marker 188
justify-content 447
justify-items 511
justify-self 511

K

kbd (tag) 142

L

label (tag) 245, 583
Landing page 36
lang (attribute) 67
lastChild 757
legend (tag) 266
let 653
letter-spacing 547
li (tag) 122, 123
line (SVG) 218
linear-gradient() 406
Line break
  <br> 113
  <wbr> 114
  preventing 116
line-height 545
Line spacing 545
Link 177
  email 183
  insert 178
  media 458
  text 177
link (tag) 78, 288
  href 80, 289

link (tag) (Cont.)
- *hreflang* 80
- *media* 80, 293
- *rel* 80
- *rel=* 213, 289
- *size* 81
- *sizes* 213
- *type* 81

Link-sensitive graphics 204, 209

List
- *designing* 557
- *graphics as bullets* 559

List display
- *changing the numbering* 124
- *description list* 128
- *nesting* 125
- *numbered* 123
- *ordered* 123
- *reversing the numbering* 124
- *unordered* 122

list-style 560

list-style-image 559

list-style-position 560

list-style-type 557

log(), JavaScript 647

Logical operator, JavaScript 669

Loop, JavaScript 672

M

main (tag) 120
- *hidden* 121

Main content <main> 120

map (tag) 204

margin (tag) 379

margin-bottom (tag) 379

margin-left (tag) 379

margin-right (tag) 379

margin-top (tag) 379

mark (tag) 146

Marker 41

Masking HTML characters 161

math (tag) 220

MathML 219

max-aspect-ratio 464

max-color 465

max-color-index 465

max-height 464

max-monochrome 465

max-resolution 465

max-width 464

Media query 457, 461

Menu bar 561

meta (tag) 85
- *charset* 92, 157
- *charset=* 86
- *content* 85, 86, 92
- *http-equiv* 86, 92
- *http-equiv=* 86
- *name* 85, 93
- *name=* 88, 89

Metadata 42, 85, 88

meta viewport 87, 467

meter (tag) 268

Microblogging 34

Microsite 36

Microsoft Editor 44

Millimeters 364

MIME type 186

min-aspect-ratio 464

min-color 465

min-color-index 465

min-height 464

min-monochrome 465

min-resolution 465

min-width 464

Model border-box 387

monochrome 465

Multiple branching, JavaScript 670

MySQL 34

N

Named colors 366

Named entity 115, 161

Naming convention 72

nav (tag) 102, 135

Navigation, flexbox 562

Navigation bar 561

new 703

nextSibling 757

nodeName 757

nodeType 757

nodeValue 757

Normalization (CSS) 602

normalize.css 602

noscript (tag) 83

not (media query) 464

null, JavaScript 661

Number 709

Numeric entity 160

O

Object 677, 709
object (tag) 228, 232
ol (tag) 123, 557
  reversed 124
  start 124
  value 125
onblur 771
onchange 772
Online magazine 33
Online store 35
only (media query) 463
onsubmit 775
optgroup (tag) 243
  label 243
option (tag) 242
  JavaScript 772
  selected 243
  value 243, 772
orientation 464
Outline 105, 107
output (tag) 256

P

p (tag) 112
padding 378
padding-bottom 379
padding-left 379
padding-right 379
padding-top 378
Paragraph text <p> 112
parentNode 756
path (SVG) 217
Percent 365
PHP 34
PHP form mailer 269
Pica 364
picture (tag) 210, 496
Pixel 364
Placeholder 260
Plain text format 41
Playing an audio file 229
Playing MP3 229
Point 364
pointer 465
polygon (SVG) 218
Portfolio website 34
position 417
  absolute 422
  fixed 426

position (Cont.)
  relative 421
  static (tag) 418
  sticky 429
Positioning 417
  absolute 422
  fixed 426
  float 434
  relative 421
  static 418, 419
POST method 271
pre (tag) 141
Preprocessor 605
preventDefault() 748
previousSibling 757
progress (tag) 267
  JavaScript 776
Protocol 74

Q

q (tag) 147
  cite 147
Query 40
querySelector() 725
querySelectorAll() 725
Query string 271

R

radial-gradient() 408
Radian 372
radius, border-bottom-right-radius 411
reCAPTCHA 185
Recommendation 414
Recordset 40
rect (SVG) 218
Reference text 177
Referencing 73
Regular expression 260
Relative URL 179
rem 539
removeAttribute() 766
removeChild() 762
removeEventListener() 740
repeat() 503
repeating-linear-gradient() 407
repeating-radial-gradient() 408
replaceChild() 763
Request 38
resolution 465
Response 38

Responsive web design 458
rgb() 368
rgba() 405
RGB mixture 368
  transparency 369
Root directory 75
Root em 365
rotate() 576
Round angle 372
Round corners 411
rp (tag) 155
rt (tag) 155
ruby (tag) 155
Ruby annotation 155

S

s (tag) 148
Safari 597
samp (tag) 142
sanitize.css 603
Sass 605
  @content 623
  @each 628
  @else 631
  @extend 618
  @for 630
  @function 632
  @import 633
  @include 615
  @media 621
  @mixin 615
  @return 632
  @while 630
  & (ampersand character) 627
  adjusting brightness 625
  color 625
  comments 634
  control structure 628
  function 632
  installing 607
  media query 621
  mixins 615
  nesting 613
  operator 624
  property nesting 614
  selector nesting 613
  variable 611
  Visual Studio Code 608
scale() 575
scope, th (tag) 168
script (tag) 82
  async 84, 645
  charset 84
  defer 84, 645
  JavaScript 641
  src 84
  type 84
SCSS → Sass
Search engine optimization (SEO) 72
section (tag) 97, 135
Section element 96
select (tag) 242
  JavaScript 772
  multiple 243
  name 243
Selection operator, JavaScript 669
Selector 283, 297
  adjacent sibling combinator 337
  attribute selector (attribute value) 312
  attribute selector (partial value) 315
  attribute selector (presence) 310
  child combinator 335
  class selector 302
  combinator 332
  descendant combinator 334
  general sibling combinator 338
  grouping 301
  ID selector 305
  negation pseudo-class 329
  pattern 298
  pseudo-class 318
  structural pseudo-class 322
  type selector 299
  universal selector 308
  user interface pseudo-classes 320
  weighting 357
Semantic HTML 130
setAttribute() 765
Shadow 554
  adding 409
Simple selector 298
skew() 577
small (tag) 155
Small caps 544
source (tag) 210, 496
  audio (tag) 230
  media 211
  sizes 211
  src 223
  srcset 211, 212
  type 211, 223
Space 115, 238

Spam ... 184
span (tag) ... 156
Specificity ... 357
Stacking ... 431
Standalone tag ... 63
Static website ... 38
stopPropation() ... 751
Strikethrough ... 552
String ... 700, 709
  JavaScript ... 658
strong (tag) ... 145
Style ... 733
style (HTML attribute) ... 286
style (tag) ... 81, 287
  media ... 82
  title ... 291
  type ... 82
sub (tag) ... 151
Sublime Text ... 45
submit ... 775
Subtitles for video and audio ... 225
summary (tag) ... 175, 275
sup (tag) ... 151
svg (tag) ... 216
  circle ... 218
  ellipse ... 218
  line ... 218
  path ... 217
  polygon ... 218
  rect ... 218
  text ... 219
SVG element, <polyline .../> ... 218
SVG format ... 214
  img (tag) ... 215
  svg (tag) ... 216
SVG tag ... 217
switch, JavaScript ... 670

T

tabindex ... 265
Table ... 163
  caption ... 571
  cell ... 166
  fixed width ... 566
  labeling ... 174
  structuring ... 169
  structuring data ... 163
table (tag) ... 164
  border ... 169
table-layout ... 567
tables ... 566
Target anchor ... 188
tbody (tag) ... 169
td (tag) ... 164
  colspan ... 166
  rowspan ... 166
template (tag) ... 768
Template string ... 660
Test ... 591
Text
  design ... 521
  indenting ... 551
  subscript <sub> ... 151
  superscript <sup> ... 151
text (SVG) ... 219
text-align ... 548
Text alignment ... 548
  vertical ... 550
textarea (tag) ... 241, 773
  cols ... 242
  maxlength ... 242
  name ... 242
  rows ... 242
  wrap ... 242
textContent ... 728
text-decoration ... 552
TextEdit ... 44
text-indent ... 551
Text markup ... 138
text-shadow ... 554
text-transform ... 544, 553
Text underline ... 151
tfoot (tag) ... 169
th (tag) ... 165
  scope ... 168
thead (tag) ... 169
time (tag) ... 151
Time data ... 372
title (tag) ... 70
tr (tag) ... 164
track (tag) ... 225
  kind ... 226
  label ... 226
  src ... 226
  srclang ... 226
Transform
  skewing ... 577
transform ... 574
Transformation ... 574
  moving ... 577
  rotating ... 576
  scaling ... 575
Transition ... 580

transition ... 580
  transition-delay ... 581
  transition-duration ... 581
  transition-property ... 580
  transition-timing-function ... 581
translate() ... 577
Transparency ... 405
true, JavaScript ... 660
typeof, JavaScript ... 657

U

u (tag) ... 148
ul (tag) ... 122, 557
undefined, JavaScript ... 661
Underline ... 552
Unicode ... 159
Unit of measurement ... 363
Universal selector
  weighting ... 358
unset ... 352
URL ... 73
User stylesheet ... 355
use strict ... 656
UTF-8 ... 86, 158, 159

V

Validating
  HTML ... 49
Validation ... 592
var ... 653
var (tag) ... 144
Vector graphic ... 214
  img (tag) ... 215
  svg (tag) ... 216
vertical-align ... 550
vh ... 541
Video
  playing ... 222
  YouTube ... 228
video (tag) ... 222
  autoplay ... 224
  controls ... 224
  height ... 224
  loop ... 224
  muted ... 224
  poster ... 224
  preload ... 224
  src ... 224
video (tag) (Cont.)
  type ... 225
  width ... 225
Viewport ... 87, 465
  height ... 365
  width ... 365
Viewport unit ... 541
visibility ... 515
Visual Studio Code ... 45
vw ... 541

W

wbr (tag) ... 114
Web app ... 37
Web browser ... 46, 592
  default stylesheet ... 355
Web browser prefix ... 413
Web crawler ... 85, 88
Web font ... 526
Web form ... 237
Weblink ... 177
Weblog ... 33
Web page
  create ... 47
Web platform ... 36
Web presence ... 32
Web server ... 40
WebVTT format ... 227
Weighting ... 357
while loop, JavaScript ... 674
Width ... 378
width ... 464, 571
  viewport ... 467
Word spacing ... 547
word-spacing ... 547
Working draft ... 414
WYSIWYG editor ... 45

X

x-height ... 364
XMLHttpRequest object ... 783

Y

YouTube ... 228

Z

z-index ... 431

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: meaganw@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://www.rheinwerk-computing.com/5695. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.rheinwerk-computing.com provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.rheinwerk-computing.com.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2023 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All products mentioned in this book are registered or unregistered trademarks of their respective companies.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.